

Universität  
Rostock



Traditio et Innovatio

Projektarbeit zum Thema:

”Optimierte Datenkodierungsalgorithmen für  
ressourcenlimitierte Umgebungen”

Betreuer:

**Dipl.-Ing. Guido Moritz**

vorgelegt dem

**Institut für Angewandte Mikroelektronik und  
Datentechnik**

der

**Universität Rostock**

Maik Gotzmann

13.12.2010

---

Thema der Projektarbeit:

## **Optimierte Datenkodierungsalgorithmen für ressourcenlimitierte Umgebungen**

Zur Überwindung der Heterogenität im Bereich der Ad-hoc Gerätekommunikation setzen sich zunehmend standardisierte Internettechnologien durch. Um die notwendige Abstraktion gegenüber der darunter liegenden Übertragungstechnologie zu gewährleisten, kommt hier vor allem das Internetprotokoll (IP) zum Einsatz. Darauf aufbauend können nun weitere Applikationsprotokolle wie HTTP, das Devices Profile for Web Services (DPWS), XMPP, uvm. umgesetzt werden, um eine Ad-hoc Kommunikation zu ermöglichen.

Zur Datenrepräsentation verwenden bestehende Anwendungsprotokolle, aus Gründen der Plattformunabhängigkeit, in der Regel XML basierte Darstellungen. Da XML aber sehr komplex ist und einen großen Overhead impliziert, ist die Kompression vor allem beim Einsatz in batteriebetriebenen, drahtlos kommunizierenden Keinstgeräten von Nutzen, um so den Energieverbrauch zu minimieren und die Lebensdauer zu erhöhen.

Im Rahmen dieser Arbeit sollen Kompressionsverfahren für XML untersucht werden. Besonderer Fokus soll auf dem standardisierten Format der Efficient XML Interchange Working Group (EXI) des W3C liegen. Ziel soll es sein, die Möglichkeit der Implementierung einer EXI kompatiblen XML-Kodierung zu evaluieren und eine erste Lösung prototypisch umzusetzen. Als Zielplattform für die Implementierung soll die TelosB Mote genutzt werden, die über einen MSP430 Mikrocontroller verfügt, der durch seine stromsparenden Eigenschaften besonders häufig in energieminimalen Systemen zum Einsatz kommt.

Folgende Aufgaben sind dabei zu lösen:

- Einarbeitung in XML und das EXI Format
- Dokumentation des EXI Formats und Bewertung der Umsetzbarkeit
- Einarbeitung in die Zielplattform TelosB
- Implementierung einer prototypischen Lösung des EXI Formats

# Inhaltsverzeichnis

|          |                                                               |           |
|----------|---------------------------------------------------------------|-----------|
| <b>1</b> | <b>Einleitung</b>                                             | <b>1</b>  |
| 1.1      | Motivation . . . . .                                          | 1         |
| 1.2      | Aufbau der Projektarbeit . . . . .                            | 2         |
| <b>2</b> | <b>Das XML Format</b>                                         | <b>3</b>  |
| 2.1      | Grundlagen . . . . .                                          | 3         |
| 2.2      | Elementare Bestandteile von XML . . . . .                     | 5         |
| 2.3      | DTD und XML Schema . . . . .                                  | 7         |
| <b>3</b> | <b>Das EXI Format</b>                                         | <b>10</b> |
| 3.1      | Grundlagen . . . . .                                          | 10        |
| 3.2      | Der EXI Header . . . . .                                      | 12        |
| 3.3      | Der EXI Body . . . . .                                        | 14        |
| 3.4      | Bewertung der Umsetzbarkeit . . . . .                         | 18        |
| <b>4</b> | <b>Implementierung des ersten Prototypen</b>                  | <b>21</b> |
| 4.1      | Konzept . . . . .                                             | 21        |
| 4.2      | Programmteil: main.cpp . . . . .                              | 23        |
| 4.3      | Programmteil: processing.cpp . . . . .                        | 23        |
| <b>5</b> | <b>Implementierung des Codegenerators</b>                     | <b>32</b> |
| 5.1      | Motivation . . . . .                                          | 32        |
| 5.2      | Implementierung . . . . .                                     | 33        |
| <b>6</b> | <b>Komplexität und Validierung der Implementierung</b>        | <b>37</b> |
| <b>7</b> | <b>Zusammenfassung und Ausblick</b>                           | <b>39</b> |
| <b>8</b> | <b>Erklärung über die selbständige Anfertigung der Arbeit</b> | <b>42</b> |

# Abbildungsverzeichnis

|     |                                                                             |    |
|-----|-----------------------------------------------------------------------------|----|
| 2.1 | Repräsentation eines XML-Dokuments als Baumstruktur . . . . .               | 3  |
| 2.2 | Beispiel einer internen Entität . . . . .                                   | 6  |
| 2.3 | Übersicht der in XSD vordefinierten Datentypen . . . . .                    | 9  |
| 3.1 | Konzept der Kommunikation XML und EXI . . . . .                             | 11 |
| 3.2 | Aufbau des EXI Headers . . . . .                                            | 13 |
| 3.3 | EXI Codierungsschema . . . . .                                              | 15 |
| 3.4 | Beispiel einer EXI Value Table . . . . .                                    | 17 |
| 3.5 | „notebook.exi“: non-Schema vs. Schema-informed . . . . .                    | 20 |
| 4.1 | Funktionsprinzip der Implementierung . . . . .                              | 22 |
| 4.2 | URI Table im Initialisierungszustand . . . . .                              | 24 |
| 4.3 | Vollständige Value Table von „notebook.exi“ . . . . .                       | 27 |
| 4.4 | XML Schema „notebook.xsd“ . . . . .                                         | 28 |
| 4.5 | Konsolenausgabe nach Decodierung von „notebook.exi“ . . . . .               | 29 |
| 4.6 | Vollständige FSM zu „notebook.xsd“ . . . . .                                | 31 |
| 5.1 | Ausschnitt aus der Datenstruktur zu <code>fragmentElements</code> . . . . . | 34 |
| 5.2 | <code>notebook.xsd</code> als fragmentierte FSM . . . . .                   | 36 |

# Tabellenverzeichnis

|     |                                                                                 |    |
|-----|---------------------------------------------------------------------------------|----|
| 2.1 | Vordefinierte interne Entitäten . . . . .                                       | 7  |
| 3.1 | Fidelity Options . . . . .                                                      | 12 |
| 3.2 | Beispiel möglicher EXI Format Versionen . . . . .                               | 14 |
| 3.3 | Übersicht aller EXI Events . . . . .                                            | 15 |
| 3.4 | Übersicht aller EXI internen Datentypen . . . . .                               | 19 |
| 7.1 | Größenvergleich zur Codierung der <code>notebook.xml</code> im EXI Format . . . | 39 |

# Abkürzungsverzeichnis

|      |   |                                      |
|------|---|--------------------------------------|
| API  | : | Application Programming Interface    |
| DPWS | : | Devices Profile for Web Services     |
| DTD  | : | Document Type Definitions            |
| EXI  | : | Efficient XML Interchange            |
| FSM  | : | Finite State Machine                 |
| IP   | : | Internet Protocol                    |
| SGML | : | Standard Generalized Markup Language |
| URI  | : | Uniform Resource Identifier          |
| W3C  | : | World Wide Web Consortium            |
| XML  | : | Extensible Markup Language           |
| XSD  | : | XML Schema Definition                |

# 1 Einleitung

## 1.1 Motivation

Das Anwendungsgebiet mobiler Kleinstgeräte wächst seit den letzten Jahren stetig an. Dies betrifft im privaten Umfeld beispielsweise Mobiltelefone und Personal Digital Assistants, als auch Sensorknoten im industriellen Umfeld. Um die Kommunikation dieser Geräte untereinander und mit zentralen Endgeräten sicherzustellen, kommt es verstärkt zur Bildung von Ad-hoc Netzwerken. Um ein Höchstmaß an Übertragungssicherheit zu gewährleisten und zur Wahrung der Interoperabilität der in diesen Netzen beteiligten Geräte, kommen in zunehmendem Maße standardisierte und bewährte Internettechnologien zum Einsatz. Als Basis wird hier meist das Internetprotokoll IP verwendet, auf dessen Grundlage dann eine Vielzahl weiterer Protokolle auf höheren Schichten aufbauen können.

Für den automatisierten Datenaustausch wird dabei insbesondere auf Webservices zurückgegriffen, da diese plattformunabhängig sind. Erreicht wird dies durch die Repräsentation der Daten als Extensible Markup Language (XML)-Dokument. Durch die Verwendung von XML entsteht jedoch ein Overhead an Daten, hervorgerufen durch deren Komplexität und Datenstruktur, der den Einsatz dieser Webservices auf Kleinstgeräten einschränkt bzw. unmöglich macht. Diese Kleinstgeräte sind dahingehend konzipiert, den Energieverbrauch zu minimieren und die Lebensdauer zu maximieren, welches in einer Reduzierung der Taktfrequenz des Mikroprozessors und eine Verringerung des Arbeitsspeichers resultiert.

Daher gibt es von Seiten des World Wide Web Consortium (W3C) Bestrebungen, ein geeignetes Kompressionsverfahren für XML-Datenstrukturen zu finden, welches den Anforderungen genügt, für die Verarbeitung der codierten Daten möglichst wenig Rechenleistung in Anspruch zu nehmen und die Datenmenge durch Repräsentation als binären Stream zu minimieren. Da bestehende Formate, wie zum Beispiel gzip und Fast Infoset, für eine Verarbeitung auf Kleinstgeräten zu rechenaufwendig sind, wurde das Efficient XML Interchange (EXI) Format entwickelt. Dieses liegt bereits jetzt als Draft Standard vor und bietet gegenüber den anderen Formaten den Vorteil, der direkten

Abarbeitung des Binärstreams. Eine vorangestellte vollständige Decodierung ist bei Verwendung des EXI Formates nicht notwendig.

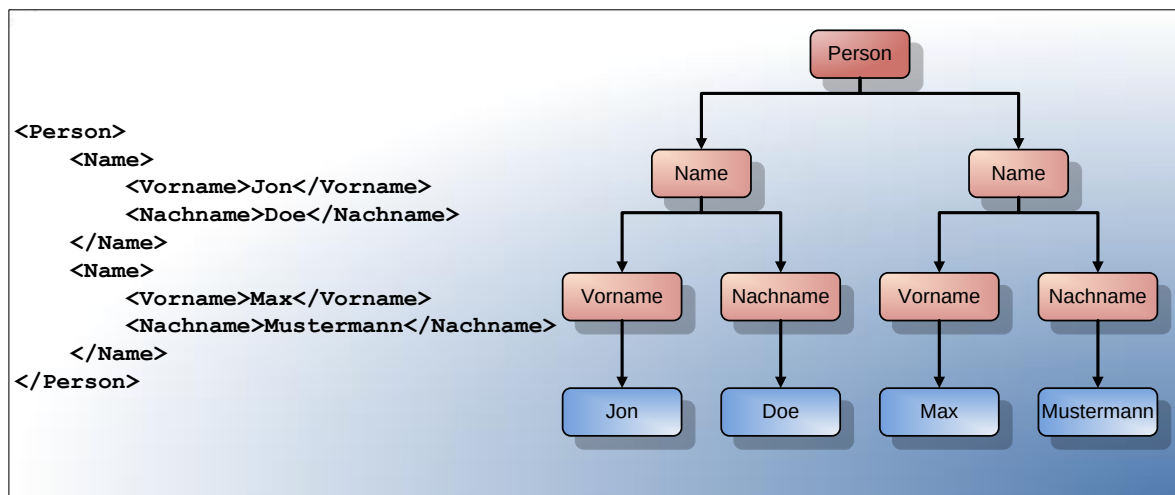
## 1.2 Aufbau der Projektarbeit

Um einen grundlegenden Einblick in den Sinn und Zweck dieser Projektarbeit zu gewinnen, wird als erstes die Spezifikation und Funktionsweise von XML inklusive der weiterführenden XML Schemas betrachtet. Basierend darauf wird im dritten Kapitel das EXI Format vorgestellt. Hier werden nicht nur die Eigenschaften und die Funktionsweise der binären Codierung zur Darstellung eines XML Streams erläutert, ebenso erfolgt hier eine erste Differenzierung und Spezialisierung für eine zukünftige Implementierung. Die Kreation eines auf den PC beschränkenden Prototypen verzweigt sich fortan in zwei Teile, die in den Kapiteln vier und fünf näher erläutert werden. Bei der ersten Implementierung handelt es sich dabei um ein rein statisches Modell zur Decodierung des Beispiel-XML-Dokumentes `notebook.xml`. Das darauf folgende Kapitel beschreibt den in JAVA implementierten Code-Generator, welcher die vorliegende erste Implementierung deutlich vereinfacht und die Verarbeitung beliebiger auf XML Schemas basierender EXI Streams automatisiert. Eine Zusammenfassung und abschließende Bewertung zu den Arbeiten an dieser Projektarbeit erfolgt schließlich im siebten und letzten Kapitel.

## 2 Das XML Format

### 2.1 Grundlagen

Die Entwicklung von XML begann 1996 unter der Schirmherrschaft des W3C, welche für die Standardisierung der Techniken zuständig ist, die das World Wide Web betreffen. XML sollte dabei auf den Erkenntnissen und Fortschritten der Standard Generalized Markup Language (SGML) aufbauen, dabei aber deutlich weniger Komplexität aufweisen[4]. Ziel war es, ein rein generisches Format zur Dokumentenauszeichnung zu entwickeln, das dem plattform- und implementierungsunabhängigen Datenaustausch zwischen Computersystemen dient und darüber hinaus vom Menschen und Computer gleichermaßen gelesen und interpretiert werden kann. 1998 wurde mit der XML Version 1.0 die erste Version standardisiert.



**Abbildung 2.1:** Repräsentation eines XML-Dokuments als Baumstruktur

Ein XML-Dokument ist dabei immer ein reines Textdokument. Daten, gleich welcher Art, werden immer als Text-Strings in das Dokument eingefügt und werden von einem Text-Markup, welches die Daten beschreibt, umgeben. Markup, dessen Name frei wählbar ist und aus dem Start- und Endtag besteht, und Daten bilden dabei ein Element. Um Elemente genauer zu bestimmen, kann auf die Verwendung von Attributen zurückgegriffen werden. Die Elemente selbst können untereinander verschachtelt

sein und ergeben einen Baum, welcher die logische Struktur eines XML-Dokumentes beschreibt. Die Blätter dieses sehr allgemeinen Baumes bilden dann die Daten ab, und an seiner Spitze steht ein einziges Wurzelelement (Root).

Die Spezifikation von XML schreibt für die Gültigkeit der Struktur feste Regeln vor, die unter dem Begriff der Wohlgeformtheit vereint werden. Damit ein XML-Dokument wohlgeformt (*engl. well formed*) ist, müssen unter anderem folgende Regeln befolgt werden:

- Jedes Starttag besitzt ein dazugehöriges Endtag
- Elemente dürfen zwar verschachtelt sein, dürfen sich dabei aber nicht überlappen
- Es gibt genau ein Wurzelelement
- Ein Element darf nicht mehrere Attribute mit gleichen Namen besitzen
- In den Daten der Elemente und Attribute sind die Zeichen `<`, `>` und `&` unzulässig

Ausnahmslos jedes XML-Dokument muss wohlgeformt sein. Bei Nichteinhaltung auch nur einer dieser Regeln, ist das entsprechende Dokument kein XML-Dokument und darf als solches auch nicht verarbeitet werden. Die Wohlgeformtheit lässt sich mit entsprechenden Editoren oder direkt auf der Homepage des W3C überprüfen.

Die Verarbeitung von XML-Dokumenten übernimmt dabei eine spezielle Softwarekomponente, die als XML-Prozessor bzw. XML-Parser bezeichnet wird. Auf unterer Ebene übernimmt er die Aufgabe des Einlesens des XML-Streams und stellt eine Programmierschnittstelle (*engl. application programming interface, kurz: API*) zur Verfügung, über die von den darüberliegenden Anwendungen Daten übergeben und empfangen werden. XML-Parser arbeiten dabei nach einem der drei grundlegenden Verarbeitungsmechanismen:

**DOM** Hier wird das XML-Dokument als Baumstruktur repräsentiert, welches der API einen unmittelbaren und wahlfreien Zugriff auf die Blätter und damit den Daten gewährt. Ebenso werden Funktionen zur Manipulationen der Baumstruktur angeboten. Die Verarbeitung anhand der Baumstruktur ist allerdings sehr speicherintensiv.

**SAX** Dabei wird das XML-Dokument als sequentieller Datenstrom repräsentiert. Tritt in ihm ein zuvor definiertes Ereignis auf, wird eine Rückruffunktion (*engl. callback function*) ausgelöst und die darüberliegende Anwendungsschicht informiert, damit die entsprechenden Daten verarbeitet werden können.

**PULL-API** Neben der sequentiellen Verarbeitung des XML-Datenstroms ähnlich der

SAX-API, bietet eine solche API auch den Zugriff auf Elemente mittels Zeiger (Iterator) an. Dadurch lässt sich die Ablaufkontrolle direkt über die Programmschicht und nicht über den Parser realisieren, welches deutlich weniger speicherintensiv ist.

XML stellt sich als äußerst flexibles Format zur Speicherung und Übertragung von Daten dar. Durch die Repräsentation von Strukturen und Daten als reine Text-Strings bleiben sie zwar ohne Umcodierung auf einer Vielzahl von Systemen les- und verarbeitbar, welches allerdings einen sehr hohen Daten-Overhead zur Folge hat. XML ist daher für die Speicherung von Daten mit vielen aufeinanderfolgenden Bitfolgen, wie sie zum Beispiel bei Bild- und Videoformaten auftreten, eher ungeeignet.

## 2.2 Elementare Bestandteile von XML

Im Folgendem sollen nun die wichtigsten Bestandteile, mit deren Hilfe sich Daten mittels eines XML Dokumentes beschreiben lassen, etwas näher betrachtet werden[2].

**Tags** Dies sind die Komponenten, welche in einem hierarchisch strukturiertem Baum die Knoten repräsentieren. Sie bestehen immer aus einem passenden Paar mit identischen und frei wählbaren Namen, wobei zwischen Groß- und Kleinschreibung (*engl. case sensitive*) unterschieden wird. Syntaktisch werden die Tag Namen dabei in `<>` eingebettet. Ein Start Tag hat dann beispielsweise die Form `<Start_Tag_Name>`, das End Tag wird zusätzlich mit `/` am Anfang des Tagnamen gekennzeichnet, z.Bsp.: `</End_Tag_Name>`. Eine Ausnahme stellen dabei leere Elemente (*engl. empty elements*) dar, die keine Daten enthalten. In diesen Fällen können Start und End Tag zu einem einzigen Empty Element Tag zusammengefasst werden. Die Kennzeichnung eines leeren Elementes erfolgt dabei mit `/` am Ende des Tag Namens, z.Bsp.: `<Tag_Name/>`. Leere Tag Namen sind in keinem Fall zulässig. Die eigentlichen Daten, die die Blätter eines Baumes repräsentieren, liegen dann zwischen Start und End Tag vor und werden als reiner Textstring dargestellt.

**Attribute** Sie versehen die Elemente mit weiteren Zusatzinformationen und sind als Schlüsselwort-Wert-Paare im Start Tag oder im Empty Element Tag vertreten und bestehen aus einem ebenfalls frei wählbaren Attributnamen und einer Wertzuweisung mittels `=`. Die ebenfalls als String dargestellten Attributwerte werden dabei von Anführungszeichen oder Hochkommata umschlossen, wie z.Bsp.: `<Datei Typ="bin">`, alternativ: `<Datei Typ='bin'>`. Um die Regeln der Wohlgeformtheit nicht zu verletzen, darf jeder Attributname nur einmal innerhalb des

Tags verwendet werden. Die Verwendung von sehr vielen Attributen führt allerdings dazu, dass das XML Dokument schnell sehr unübersichtlich wird und damit einen Teil seiner Lesbarkeit einbüßt. Einen Vorteil stellt die Verwendung von Attributen gegenüber den in den Elementen gespeicherten Daten jedoch bei rechentechnischer Verarbeitung dar, da die Attributwerte von XML Prozessoren oft in Hash-Tabellen gespeichert werden. Der Zugriff auf die in ihr gespeicherten Werte erfolgt dabei direkt und ist schneller, als den Strukturbaum zu durchlaufen.

```
<!ENTITY version "1.25">
<programm>
    <version>&version;</version>
</programm>
```

**Abbildung 2.2:** Beispiel einer internen Entität

**Interne Entitäten** Sie stellen Speichereinheiten dar und sind vergleichbar mit der `#define` Anweisung aus der Programmiersprache C. Die Deklaration einer Entität erfolgt innerhalb des XML Dokumentes und beginnt mit dem Schlüsselnamen `!ENTITY`, gefolgt von einem Namen und einem in Anführungszeichen oder Hochkommata eingeschlossenem Wert. Die Referenz auf diesen Wert erfolgt dann innerhalb eines Elements durch ein vorangestelltes Ampersand, gefolgt vom Namen der Entität mit abschließendem Semikolon, wie z.Bsp. in Abbildung 2.2. Im XML Standard existieren auch bereits vordefinierte Entitäten, die dazu dienen, unzulässige Zeichen in den Werten der Elemente und Attribute darzustellen. Eine Auflistung dieser vordefinierten internen Entitäten kann der Tabelle 2.1 entnommen werden. Die Referenzauflösung bei Auswertung des XML Dokumentes übernimmt dabei der XML Prozessor.

**Externe Entitäten** Bei Verwendung dieser werden Abschnitte des ursprünglichen XML Dokumentes ausgelagert und in physikalisch separate XML Dokumente gespeichert. Dadurch werden sehr große und umfangreiche XML Dokumente besser lesbar. Die Deklaration erfolgt auch in diesem Fall mit dem Schlüsselwort `!ENTITY`, gefolgt von einem Namen. Danach verweist das Schlüsselwort `SYSTEM` auf die verknüpfte externe Entität, also auf das physikalisch ausgelagerte XML Dokument. Das Einfügen des XML Fragmentes in das Stammdokument erledigt auch hier der XML Prozessor nach Erkennen der entsprechenden Referenz in einem Element.

**Namensräume** Um Namenskonflikte innerhalb eines XML Dokumentes zu vermeiden,

| Zeichen | Referenz                | Beschreibung      |
|---------|-------------------------|-------------------|
| <       | <code>&amp;lt;</code>   | kleiner als       |
| >       | <code>&amp;gt;</code>   | größer als        |
| &       | <code>&amp;amp;</code>  | ampersand         |
| '       | <code>&amp;apos;</code> | Apostroph         |
| "       | <code>&amp;quot;</code> | Anführungszeichen |

**Tabelle 2.1:** Vordefinierte interne Entitäten

wurden mit der zweiten Ausgabe der XML Spezifikation 1.0 Namensräume (*engl. namespaces*) eingeführt. Die Deklaration der Namensräume erfolgt noch vor den betreffenden Elementen innerhalb eines Start Tags mit dem Schlüsselwort `xmlns`. Ein Doppelpunkt trennt den nun folgenden Namen für den Namensraum vom Schlüsselwort ab. Anschließend erfolgt die eindeutige Zuweisung dieses Namens zu einem Uniform Resource Identifier (URI). Alle folgenden Elemente und Attribute können nun durch Voranstellen des Namens für den Namensraum, gefolgt von einem Doppelpunkt eindeutig einer URI zugewiesen werden. Durch Weglassen des Doppelpunktes und des Namens nach ihm bei der Deklaration des Namensraumes, wird für alle nachfolgenden Elemente und Attribute ein neuer Default Namensraum definiert. Dieser ist dann für alle Elemente und Attribute gültig, die fortan ohne Namespace Präfix aufgeführt sind.

## 2.3 DTD und XML Schema

Die Möglichkeit, Attribute und Elemente innerhalb eines XML Dokumentes frei wählen und kombinieren zu können, macht XML zu einem äußerst flexiblen Werkzeug zur Entwicklung eines eigenen Formates. Anwendungen, die die in den XML Dokumenten enthaltenen Daten jedoch verarbeiten sollen, sind auf die Einhaltung entsprechender Regeln angewiesen, da sie nur ein bestimmtes XML Format zuverlässig verarbeiten können. Um schon bei der Erstellung eines XML Dokumentes sicherzugehen, dass es von der zu verarbeitenden Anwendung als konform angesehen wird, wurden Document Type Definitions (DTD) und Schemas eingeführt. Mit ihnen existiert ein Regelwerk, welches vorschreibt, welche Elemente und Attribute unter welchen Zuständen in einem XML Dokument existieren dürfen. Der XML Prozessor kann so im Vorfeld der weiteren Verarbeitung eine Überprüfung vornehmen, ob das vorliegende XML Dokument sich an die in dem Schema vorgegebenen Regeln hält. Dieser Vorgang wird Validierung genannt und ist ein optionaler Schritt, der bei Gültigkeitsverletzung, anders als bei Verstoß gegen die Wohlgeformtheit, nicht zwangsläufig zum Abbruch der XML Verarbeitung führen muss.

DTDs können innerhalb des XML Dokumentes oder mittels Verweis auf eine externe Ressource ähnlich einer externen Entität definiert werden und folgen einer formalen Syntax, die dabei nicht der hierarchischen Struktur von XML folgt. Eine DTD listet alle Elemente, Attribute und Entitäten und deren kontextabhängigen Verwendung auf, die im betreffenden XML Dokument Verwendung finden. Alle im Dokument auftretenden Elemente müssen in der DTD auch deklariert werden, wobei auch Deklarationen erlaubt sind, die im XML Dokument gar nicht vorkommen. Dieses Regelwerk folgt dem Prinzip, das alles verboten ist, was nicht erlaubt ist. Entspricht das Dokument in seiner Gesamtheit den Deklarationen der DTD, so wird es gültig genannt. Durch die Verwendung von DTDs werden der Form, die das XML Dokument anzunehmen hat, zwar Beschränkungen auferlegt, dennoch verbleiben innerhalb dieser Grenzen eine gewisse Flexibilität. Eine DTD trifft keine Aussage zu Länge, Typ, Struktur oder Bedeutung der in den Elementen und Attributen verwendeten Werte.

Da für viele Anwendungen aber eine mächtigere und ausdrucksstärkere Validierungsmöglichkeit erforderlich ist, wurde die XML Schema Definition (XSD) entwickelt. Sie ist als direkter Nachfolger der DTD anzusehen und hat diese auch weitgehend verdrängt. Mit Hilfe der XSD werden einem XML Dokument komplexe Beschränkungen hinsichtlich der Verwendung von Elementen und Attributen auferlegt.

Dabei ist die Deklaration der XSD wiederum ein eigenständiges XML Dokument, welches ebenfalls durch den XML Prozessor ausgewertet werden kann. Es besteht die Möglichkeit, innerhalb einer XSD weitere Schemas zu importieren und somit zu kombinieren, was der Wiederverwendung einzelner Schemas zu Gute kommt. Ebenso kann das XML Dokument durch mehrere Schemas instantiiert werden.

Ein großer Vorteil der XSD gegenüber DTD stellt auch das Vorhandensein einer Vielzahl von vordefinierten Datentypen dar[5]. So existieren in der Spezifikation 19 primitive Typen, aus denen dann weitere abgeleitete Datentypen zur Verfügung stehen. Abbildung 2.3 gibt einen Überblick über alle in XSD bereits vordefinierten Datentypen. Aus den so vorhandenen Typen kann der Nutzer weitere primitive Datentypen durch Ableitung selbst definieren. Die Definition von komplexen Datentypen eröffnet dabei die Möglichkeit, Strukturen aus mehreren Elementen und Attributen zusammenhängend zu definieren.

Die Verwendung von DTD und XSD erleichtert den Programmierern von Anwendungen, welche die Daten der XML Dokumente verarbeiten, die Komplexität und Flexibilität von XML, bezogen auf den Zweck und Einsatz, einzuschränken. An den durch die String-Repräsentation hervorgerufenen Daten-Overhead ändert ihr Einsatz allerdings nichts.

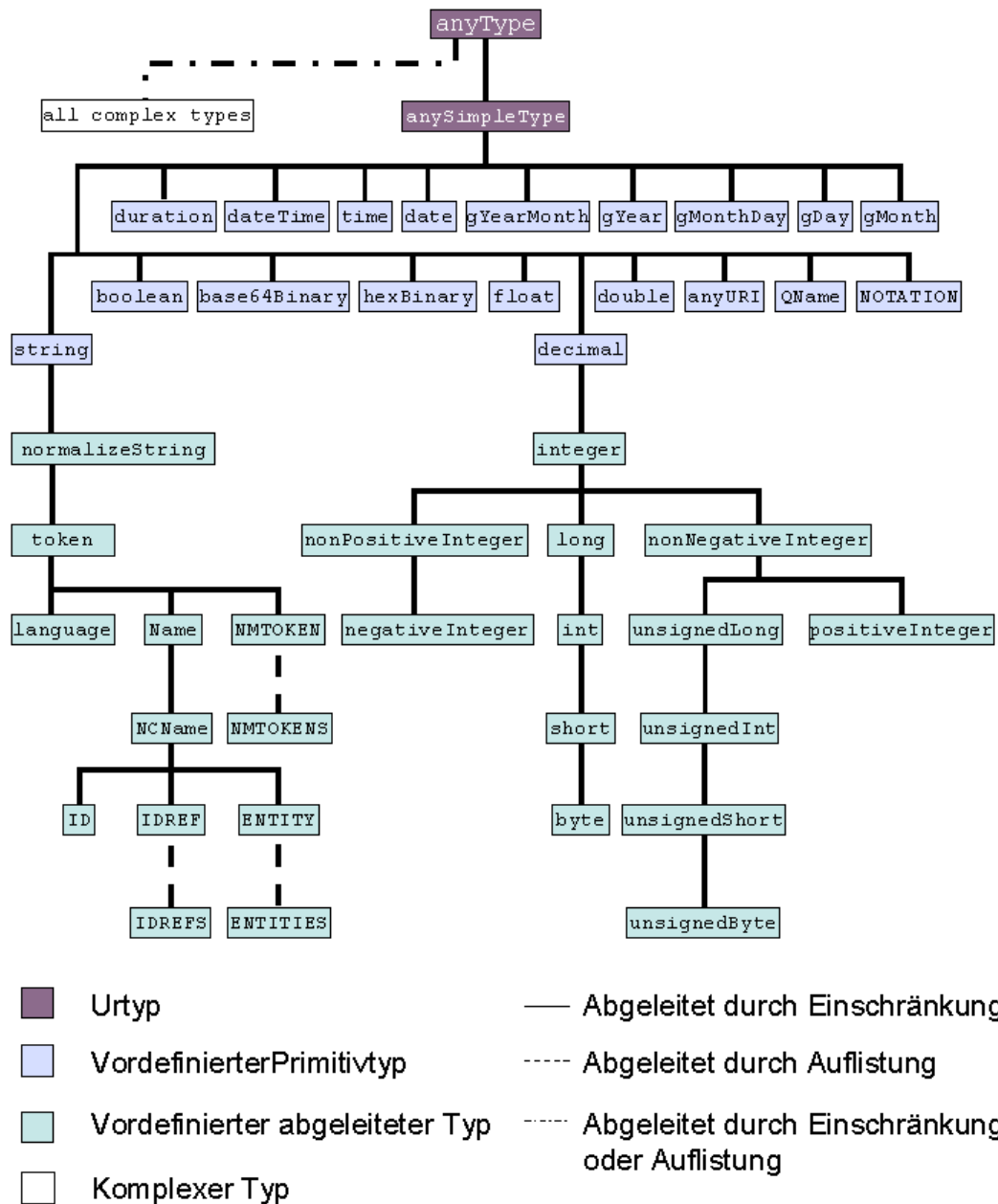


Abbildung 2.3: Übersicht der in XSD vordefinierten Datentypen  
 Quelle: <http://www.w3.org/TR/xmlschema-2/>

## 3 Das EXI Format

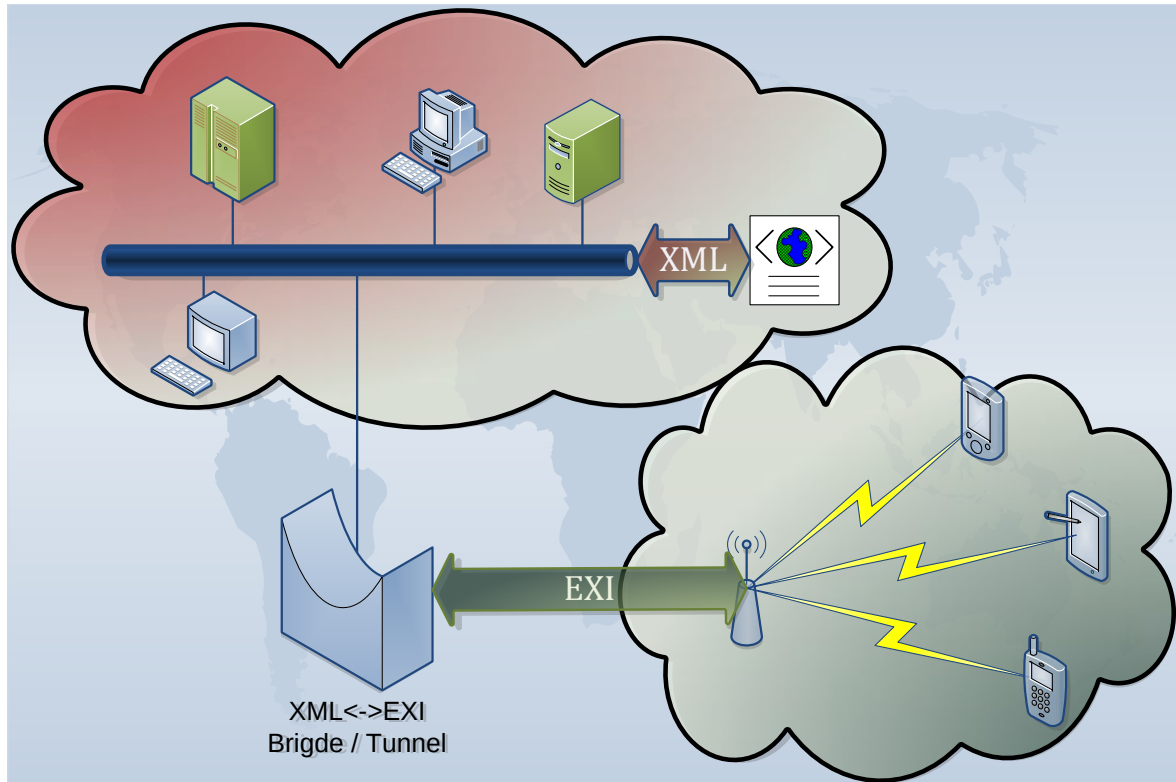
### 3.1 Grundlagen

Um einen einzelnen boolschen Wert zu übertragen oder zu speichern, wird bei Verwendung eines binären Formates im optimalen Fall lediglich ein einzelnes Bit benötigt. Die gleiche Darstellung mit Hilfe eines XML Dokumentes benötigt dagegen eine Vielzahl von Bytes. So ist es nicht verwunderlich, dass mit Einführung von XML als Standard viele Anstrengungen unternommen wurden, XML Dokumente mittels binären Datenstroms eindeutig zu repräsentieren. Die so codierte Darstellung ermöglicht es, den Energieverbrauch, den Speicherplatzbedarf und die erforderliche Bandbreite für den Einsatz auf Kleinstgeräten zu minimieren.

Erste Ansätze brachten Formate wie XML+gzip, Fast Infoset, Fujitsu Binary, Xebus und esXML hervor. Die durch sie erzielte Kompaktheit der codierten Streams ist allerdings sehr unstetig und teilweise unbefriedigend[7]. Durch die XML Binary Characterization Working Group des W3C wurden Anforderungen an das zu entwickelnde Format definiert, die durch die bereits vorhandenen Formate jedoch nicht erfüllt werden. Die wichtigsten Anforderungen an ein zukünftiges binäres Format zur Repräsentation von XML sind das direkte Lesen- und Schreiben im Datenstrom, die Kompaktheit und eine effiziente Verarbeitung des Datenstroms[6]. Die im Jahr 2005 gegründete Efficient XML Interchange Working Group begann mit der Entwicklung des EXI Formates, welcher derzeit als Draft Standard vorliegt und seit Dezember 2009 der W3C als Kandidat zur Standardisierung vorgeschlagen wurde[8].

Mit Entwicklung des EXI Formates wurden Konzepte der Formalen Sprachtheorie und der Informationstheorie zusammengefasst und ein relativ einfacher Algorithmus zur Codierung gefunden, der dem Format eine breite Zugänglichkeit bei hoher Flexibilität und hoher Performance ermöglicht. Dabei benutzt der Algorithmus eine spezielle Grammatik, um an jedem Punkt innerhalb des XML Dokumentes eine Vorhersage zu treffen, was als nächstes auftreten wird. Dies ermöglicht dem Algorithmus die Struktur eines XML Dokumentes in Codes, den sogenannten **Event Codes**, variabler Länge zu codieren. Das Event, welches mit höchster Wahrscheinlichkeit als Alternative auftreten

könnte, wird mit dem Code geringster Bitlänge versehen. Die Codierung der Event Codes ist denen der Huffman Codierung recht ähnlich, ohne jedoch einen vergleichbaren Aufwand an Rechenarbeit und Speicherplatz zu betreiben.



**Abbildung 3.1:** Konzept der Kommunikation XML und EXI

Das EXI Format bietet Möglichkeiten an, bestimmte Events schon beim Encoding herauszufiltern und zu unterdrücken, um die Übertragung redundanter oder unerwünschter Informationen zu unterbinden. So kommt es beispielsweise unter Verwendung von SOAP zu keiner Verarbeitung von Kommentaren und DTDs. Diese Filtermöglichkeiten, welche in der Tabelle 3.1 aufgelistet sind, werden durch die **Fidelity Options** definiert.

Bei der Speicherung des binären EXI Streams wird zwischen **Bit** und **Byte Alignment** unterschieden. Diese Unterscheidung gibt an, in welcher Form die Codes im binären Datenstrom abgelegt werden. Bit aligned bedeutet dabei, dass jeder Code eben nur die entsprechende Anzahl an Bits im Datenstrom verbraucht, wie er eben benötigt. Bei der Speicherung im Byte aligned Modus wird jedem Code ein separates Byte zur Verfügung gestellt, nicht benötigter Speicherplatz wird mit Padding Bits aufgefüllt. In der Praxis sollte jedoch der Bit aligned Methode zur Darstellung des EXI Streams Vorrang gewährt werden, da durch ihr deutlich kleinere binäre Streams zustande kommen. Die Verwendung von Byte aligned Streams bietet sich als Werkzeug zur Fehlersuche

| Option                              | Wirkung                                                                            |
|-------------------------------------|------------------------------------------------------------------------------------|
| <code>Preserve.comments</code>      | Filtert Kommentare aus dem XML Dokument, diese erscheinen nicht im EXI Stream      |
| <code>Preserve.pis</code>           | Verhindert die Verarbeitung von Processing Instructions                            |
| <code>Preserve.dtd</code>           | Keine Verarbeitung von DTD Deklarationen                                           |
| <code>Preserve.prefixes</code>      | Strings der Prefixe zu den Namensräumen werden nicht im EXI Stream gespeichert     |
| <code>Preserve.lexicalValues</code> | Verhindert die Nutzung der in EXI eingebauten Datentypen, reine String Darstellung |

Tabelle 3.1: Fidelity Options

eher nur in der Entwicklungsphase der Software an, die die Verarbeitung von EXI Streams auf dem Zielsystem durchführt.

Um die Größe des EXI Streams noch weiter zu reduzieren, bietet das EXI Format die Möglichkeit an, eine konventionelle Komprimierung auf ihn anzuwenden. Dabei besteht die Auswahl zwischen pre-compression und die Verwendung des DEFLATE Compressed Data Format zur Verfügung. Bei letztgenannter Komprimierung wird zuerst der Byte aligned EXI Stream durch den Encoder erzeugt. Über ihn als Ganzes erfolgt dann die Komprimierung durch den DEFLATE Algorithmus, welcher eine Kombination aus LZ77 Algorithmus und Huffman Codierung ist. Der Zugewinn an Kompaktheit wird dabei durch einen nicht unerheblichen Rechenaufwand zur Komprimierung/Dekomprimierung erkauft. Daher bleibt die Nutzung dieses Features im Rahmen dieser Projektarbeit auch unbeachtet, da sich der Einsatz dieser Komprimierungsmethoden auf Kleinstgeräten nicht rechnet.

## 3.2 Der EXI Header

Wie fast jedes Format, besitzt auch EXI einen eigenen Header, der weitere Informationen zur Verarbeitung der im EXI Stream vorhandenen Daten bereit hält. Die Aufgabe des EXI Headers ist es, den vorliegenden Datenstrom als einen im EXI Format Gehaltenen zu identifizieren und ihn abzugrenzen von einem normalen XML-Dokument. Weiterhin sind im Header Informationenn über die verwendete Version des Formates und Optionen, unter deren Umstände der vorliegende Stream entstanden und zu bearbeiten ist, enthalten. Eine schematische Darstellung des EXI Headers kann der Abbildung 3.2 entnommen werden.

**EXI Cookie** Die W3C empfiehlt die Verwendung des EXI Cookies, er ist also optional, und deren Verwendung wird nicht explizit vorgeschrieben. Das aus vier ASCII

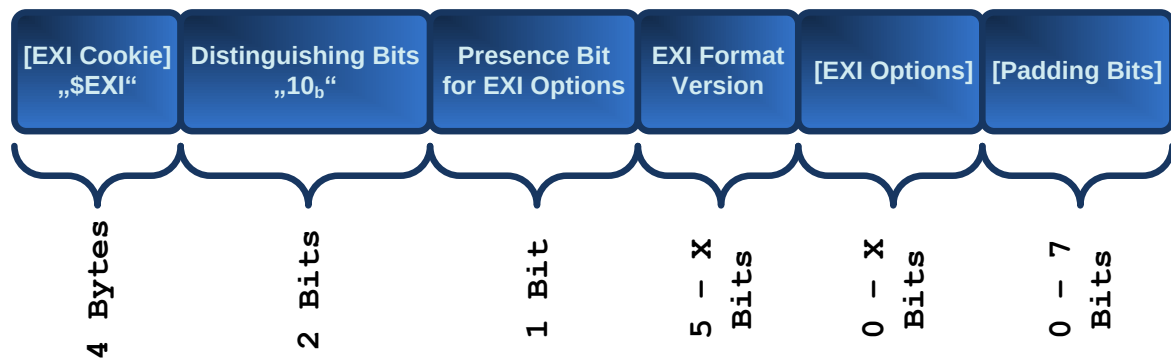


Abbildung 3.2: Aufbau des EXI Headers

Zeichen zu je einem Byte bestehende Wort **\$EXI** am Anfang des EXI Headers soll den vorliegenden Stream eindeutig als einen im EXI Format Vorliegenden identifizieren. Dadurch soll er sich deutlich von den vielen anderen im World Wide Web vertretenen Übertragungsformaten unterscheiden.

**Distinguishing Bits** Diese aus den zwei Bits bestehende Bitfolge: 10 ist im Gegensatz zum EXI Cookie von der Format Spezifikation dringend vorgeschrieben und muss auf alle Fälle verwendet werden. Sie dient der eindeutigen Unterscheidung, ob es sich bei den vorliegenden Datenstrom um ein XML Dokument oder einen EXI Stream handelt. Gemäß den Spezifikationen zum XML Format der Versionen 1.0 und 1.1 darf dort eben kein Zeichen an erster Stelle, ganz gleich welcher Zeichensatz dabei benutzt wurde, auftauchen, deren erste zwei Bits dieser Bitfolge entsprechen. Gemäß den Forderungen der Wohlgeformtheit eines XML Dokumentes, hat ein XML Prozessor bei Auftauchen dieser ersten beiden Bits in einem Dokument seine Arbeit sofort abubrechen. Eine Abgrenzung zu anderen binären Streams ist aber mit Hilfe dieser beiden Distinguishing Bits nicht möglich.

**Presence Bit for EXI Options** Dieses einzelne Bit gibt das Vorhandensein von EXI Format Optionen an. Eine 0 steht dabei für das Nichtvorhandensein dieser, eine logische 1 erklärt, dass EXI Optionen deklariert wurden. Die Präsenz bzw. Absenz dieser anzugeben ist in jedem EXI Stream Pflicht.

**EXI Format Version** Dieser Teil des Headers gibt Aufschluss darüber, mit welcher Format Version der vorliegende EXI Stream encodiert wurde. Mit ihm ist es möglich, zwischen Absender und Empfänger des EXI Streams eine gewisse Toleranzschwelle des zu verarbeitenden Formates festzulegen. Eine logische 0 am Anfang dieser Header Sektion erklärt, dass es sich bei dem vorliegendem EXI Stream um Einen der Spezifikation 1.0 Final Version handelt und alle darin spezifizierten Funktionen unterstützt. Eine logische 1 am Anfang, definiert eine Preview Version mit eingeschränkter Funktionalität. Anschließend folgen eine bis mehrere 4-Bit

umfassende Gruppen von **Unsigned Integer** Werten, die die tatsächliche Versionsnummer angeben. Eine Gruppe davon ist ausreichend, sofern der Wert der eingelesenen Gruppe sich im Bereich von 0-14 bewegt. Die Versionsnummer ergibt sich dann aus diesem Wert zzgl. einer Eins. Liegt die Versionsnummer oberhalb der Wertigkeit von 15, so wird ein weiteres Quartett an Bits eingelesen und sie mit den vorhergehendem binären Wert addiert. Auch hier wird der resultierende Wert um eins erhöht und ergibt dann die Versionsnummer. Eine kurze Übersicht möglicher Versionsnummern ist der Tabelle 3.2 zu entnehmen.

| Bit-Folge   | Version           |
|-------------|-------------------|
| 1 0000      | Preview version 1 |
| 0 0000      | Final version 1   |
| 0 1110      | Final version 15  |
| 0 1111 0000 | Final version 16  |
| 0 1111 0001 | Final version 17  |

**Tabelle 3.2:** Beispiel möglicher EXI Format Versionen

**EXI Options** Diese Sektion innerhalb des EXI Headers gibt Aufschluss darüber, wie die im EXI Stream vorliegenden Daten zu verarbeiten sind. Hier werden Informationen zu den verwendeten Fidelity Options sowie weitere Codierungseinstellungen übermittelt. Die Angabe dieser Optionen ist von Seiten der W3C nicht zwingend vorgeschrieben. Die Optionen werden dabei als ein eigenständiges XML Dokument dargestellt, welches mit Hilfe eines entsprechenden XML Schemas innerhalb des EXI Streams encodiert wird.

**Padding Bits** Diese entfallen bei Erstellen des EXI Streams im Bit aligned Modus. Das gerade angefangene Byte wird jedoch mit Paddingbits der Wertigkeit 0 aufgefüllt, sofern der Byte aligned Modus oder eine der beiden konventionellen Komprimierungsmethoden benutzt wird.

### 3.3 Der EXI Body

Der EXI Body enthält das gesamte XML Dokument in codierter Form. Hierbei kommt es zur Unterscheidung zwischen structure coding und content coding. Durch das **structure coding** wird die Struktur des ausgehenden XML Dokumentes beschrieben. Durch die EXI eigene Grammatik wird jedem Event des XML Dokumentes ein eindeutiger EXI Event Code zugeordnet. Das **content coding** platziert die Daten der Elemente und Attribute an entsprechender Stelle zwischen den Eventcodes in den EXI Stream.

Die Codierung dieser Daten erfolgt dabei im einfachsten Fall als String-Repräsentation. Abbildung 3.3 stellt den Ablauf der Codierung schematisch dar.

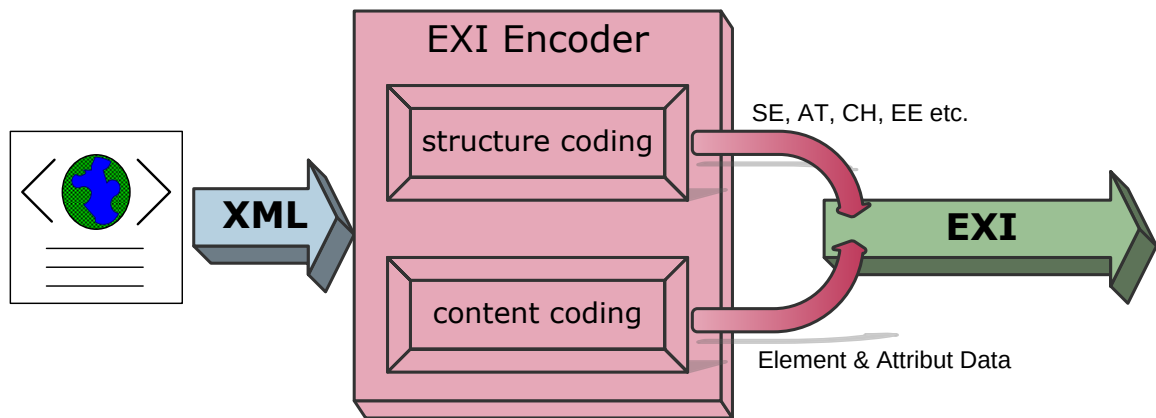


Abbildung 3.3: EXI Codierungsschema

Tabelle 3.3 zeigt alle in EXI verwendeten Events, die in ihrer Gesamtheit die EXI Grammatik bilden. So wird der Start Tag eines XML Elements mit **Start Element SE**, das korrespondierende Event auf den End Tag als **End Element EE**, bezeichnet. Die **Attribute** eines Starttags werden mit **AT** notiert. Das Event, auf das die Daten eines Elementes folgen, werden mit **Character CH** bezeichnet. Die Events SE und AT können dabei in drei unterschiedlichen Variationen auftreten, da sich deren Tag Name bzw. Attributname neben dem eigentlichen Namen, den **qname**, auch aus einer URI zusammensetzen kann.

| EXI Event              | Event Inhalt                  | Grammatik |
|------------------------|-------------------------------|-----------|
| Start Document         |                               | SD        |
| End Document           |                               | ED        |
| Start Element          | qname                         | SE(qname) |
|                        |                               | SE(*)     |
|                        |                               | SE(uri:*) |
| End Element            |                               | EE        |
| Attribute              | qname, value                  | AT(qname) |
|                        |                               | AT(*)     |
|                        |                               | AT(uri:*) |
| Characters             | value                         | CH        |
| Namespace Declaration  | uri, prefix, local-element-ns | NS        |
| Comment text           |                               | CM        |
| Processing Instruction | name, text                    | PI        |
| DOCTYPE                | name, public, system, text    | DT        |
| Entity Reference       | name                          | ER        |
| Self Contained         |                               | SC        |

Tabelle 3.3: Übersicht aller EXI Events

Jeder Event-Typ wird durch seinen Eventcode dargestellt, der sich aus einer Codegruppe von einem bis drei einzelnen Codes zusammensetzt. Für jedes neu auftretende Event während der Abarbeitung des EXI Streams wird ein Standardsatz der Eventcodes gebildet, so dass jedem auftretendem Event ein eindeutiger Code zugeordnet ist. Jeder Codesatz wird in Form eines Stacks verwaltet. Durch PUSH und POP Operationen auf diesen Stack ändert sich die Anzahl der Event Codes im Zuge der Verarbeitung. Die notwendige Bitlänge  $l$  eines im Stream enthaltenen Eventcodes ergibt sich dann aus dem *logarithmus dualis* ( $\lg$ ) des größten Codes mit der Wertigkeit  $m$ , aufgerundet auf die nächste ganze Zahl gemäß der Formel 3.1. Die Anzahl der verwendeten verschiedenen Eventcodes lässt sich durch entsprechendes Setzen der Fidelity Options reduzieren. Durch Herausfiltern bestimmter Eventtypen, wie beispielsweise Kommentar oder Processing Instructions, wird diesen auch kein Eventcode zugeordnet. Dadurch reduziert sich die Anzahl der zu verwaltenden Codegruppen, und die Bitlänge jedes einzelnen Codes verringert sich.

$$l = \lceil \lg(m) \rceil \quad (\text{m: Anzahl/Wertigkeit}) \quad (3.1)$$

Um bei sich wiederholten Auftreten eines identischen Strings Speicherplatz innerhalb des EXI Streams zu sparen, werden durch einen EXI Prozessor die folgenden vier String Tabellen verwaltet:

- **URI**
- **Prefix**, speichert die Verknüpfung zur entsprechenden URI
- **Local-Name**, enthält Tag- und Attributnamen
- **Value**, enthält die Daten aus den Elementen und Attributen

Dadurch ist es möglich, einen bereits aufgetretenem String einen kompakten Code zuzuordnen. Tritt somit ein String zum wiederholten mal im XML Dokument auf, welches z.Bsp. bei Tag Namen immer der Fall ist, so wird auf diesen im EXI Stream durch diesen Code referenziert. Bis auf die URI Tabelle, sind alle Tabellen zu Beginn eines EXI Streams leer. Während des Encodings wird bei Auftreten eines Strings jedesmal die entsprechende String Tabelle auf das Vorhandensein dieses Strings hin überprüft. Existiert der String noch nicht, so wird die entsprechende Tabelle um einen Eintrag erweitert, und er wird dort eingetragen. Dabei wird diesem String eine laufende Nummer, seinem Identifizierer, entsprechend der Auftrittsreihenfolge zugeordnet. Die Bitlänge  $l$  des Identifizierers ergibt sich dann nach Formel 3.1, wobei  $m$  die Anzahl der Einträge in der jeweiligen Tabelle zum Zeitpunkt des Auftreten des Strings entspricht. Das EXI Format sieht bei der Codierung der Identifizierer der Local-Name und Value Table zwei

Arten des Zugriffs vor. Die **globale Codierung** des Identifizierers gibt die absolute Position innerhalb der Table an. Seine Bitlänge ergibt sich aus der aktuellen Anzahl aller Einträge in der Table. Bei Zugriff durch **lokaler Codierung** erfolgt eine Partitionierung der String-Table. So erfolgt bei der Local-Name Table neben der Speicherung des **qnames** auch die Speicherung der Information, zu welchem Namespace ein jeder gehört. Bei der Value Table wird zusätzlich zum Wert auch die Zugehörigkeit dessen zum Element gespeichert. Bei wiederholten Zugriff wird zuerst die Zusatzinformation ausgewertet und der lokale Identifizierer zeigt dann auf den String innerhalb der Partition. Dadurch verringert sich die Bitlänge des Identifizierers deutlich. Die Abbildung 3.4 zeigt die Value Table am Beispiel der „notebook.xml“ nach vollständiger Codierung[9].

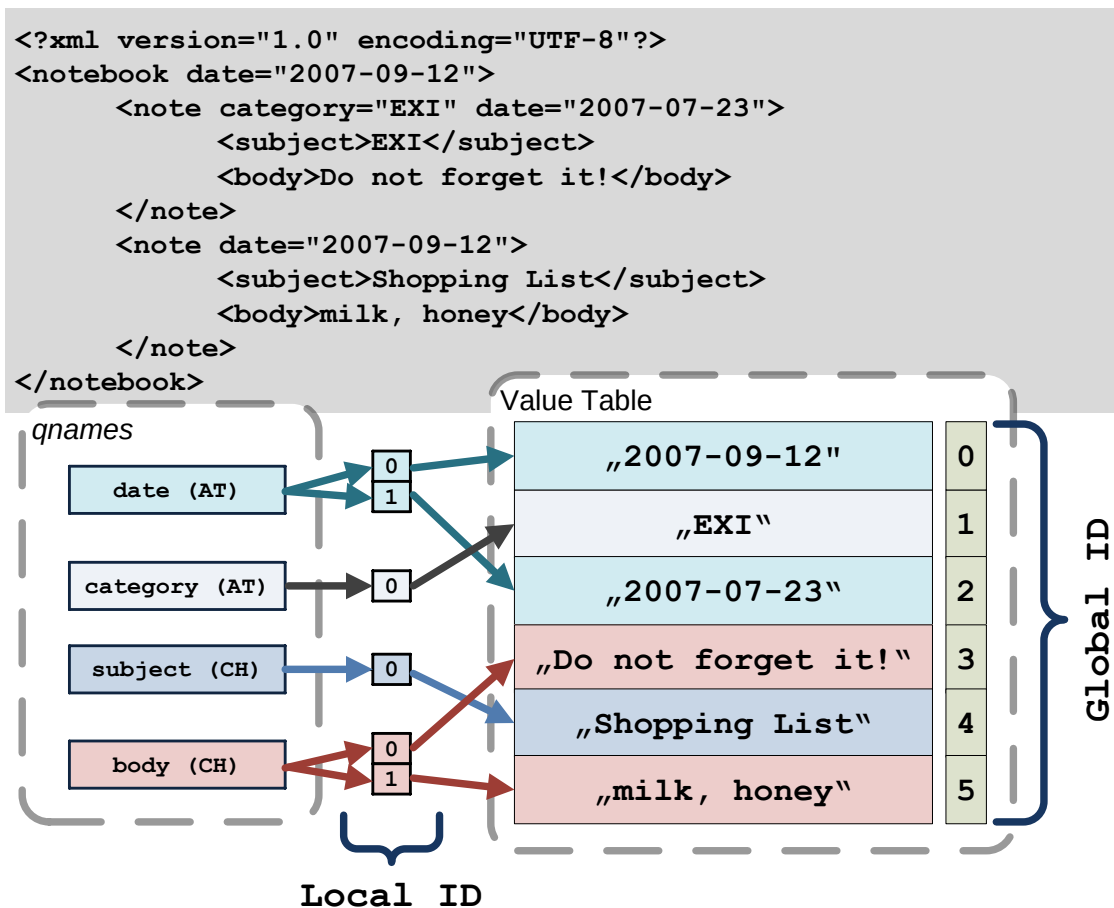


Abbildung 3.4: Beispiel einer EXI Value Table

Alle bis dahin getätigten Aussagen zum EXI Stream beziehen sich in erster Linie auf die Codierung von XML Dokumenten ohne entsprechendem Schema (**non-Schema informed**). Unter Einbeziehung von XML Schema zur Codierung (**Schema informed**) ergeben sich aber einige Unterschiede. Durch die Analyse des entsprechenden XML Schemas vor dem Encoding wird eine eigene spezielle Grammatik definiert, die der Co-

dierung ein starres Modell vorschreibt. Durch Kenntnis des Schemas ist dem Encoder an jeder Stelle bekannt, was als nächstes Event eintreten kann. Folglich müssen die vorgehaltenen Eventcodes nicht das gesamte Spektrum möglicher Events umfassen, welches wieder in einer kürzeren Bitlänge der einzelnen Eventcodes resultiert. Durch den Aufbau der eigenen Grammatik ist es nun auch nicht mehr nötig, Tag- und Attributnamen in den EXI Stream zu speichern. Damit entfällt prinzipiell die Verwaltung der Local-Name Table. Sofern keine Abweichungen vom XML Schema akzeptiert werden, die Verarbeitung wird **strict mode** genannt, entspricht die generierte Grammatik einer Finite State Machine (FSM). Die möglichen Events und ihre Codes werden dabei lexikographisch gemäß ihrer `qnames` gefolgt von der zugeordneten URI sortiert. Die Verarbeitung von EXI Streams im **non-strict mode** sieht dabei jeweils den weiteren Eventcode **Undeclared** als Letzten vor, um eben auf Abweichungen vom Schema zu reagieren.

Durch die Codierung mit Hilfe der XML Schema ist ebenso auch die Nutzung aller dem Schema zur Verfügung stehenden Datentypen, inklusive der selbst definierten komplexen Typen, für Elemente und Attribute möglich. Dazu sind in EXI selbst ein Mindestmaß an Datentypen bereits eingebaut, um das gesamte Spektrum der XSD Datentypen abzudecken. Diese typisierten Daten werden fortan auch nicht mehr als reine Strings innerhalb des EXI Streams repräsentiert, sondern spezifisch nach Datentyp durch spezielle Algorithmen codiert. Dadurch wird zum einen die Kompaktheit des EXI Streams weiter gesteigert, zum anderen können die Daten deutlich schneller und einfacher ausgewertet und verarbeitet werden. Tabelle 3.4 zeigt eine Übersicht, der in EXI vordefinierten Datentypen und die jeweilige Äquivalenz zu den Typen der XSD.

## 3.4 Bewertung der Umsetzbarkeit

Mit Verwendung des EXI Formates, steht ein sehr leistungsfähiges und flexibles Werkzeug zur eindeutigen binären Repräsentation von XML Dokumenten zur Verfügung. Die Kompaktheit der codierten EXI Streams bleibt im Vergleich zu konkurrierende Verfahren unerreicht[3].

Die Codierung anhand der reinen XML Instanz ohne Verwendung von XSD führt zwar zu einem Höchstmaß an Flexibilität, doch die Verwaltung der Eventcodesätze und String Tables ist zum einen im gewissen Maß immer noch recht speicherintensiv. Zum anderen verlangt die Implementierung einiges an programmiertechnischen Aufwand ab, um alle Funktionen des Formates nutzen zu können. Die Kompaktheit des EXI Streams wird im non-Schema informed Modus durch die Repräsentation der Daten als reine

| EXI Datentyp           | EXI Datatype ID               | Äquivalenter XSD Datentyp                      |
|------------------------|-------------------------------|------------------------------------------------|
| Binary                 | <code>exi:base64Binary</code> | base64Binary                                   |
|                        | <code>exi:hexBinary</code>    | hexBinary                                      |
| Boolean                | <code>exi:boolean</code>      | boolean                                        |
| Date-Time              | <code>exi:dateTime</code>     | dateTime                                       |
|                        | <code>exi:time</code>         | time                                           |
|                        | <code>exi:date</code>         | date                                           |
|                        | <code>exi:gYearMonth</code>   | gYearMonth                                     |
|                        | <code>exi:gYear</code>        | gYear                                          |
|                        | <code>exi:gMonthDay</code>    | gMonthDay                                      |
|                        | <code>exi:gDay</code>         | gDay                                           |
|                        | <code>exi:gMonth</code>       | gMonth                                         |
| Decimal                | <code>exi:decimal</code>      | decimal                                        |
| Float                  | <code>exi:double</code>       | float, double                                  |
| Integer                | <code>exi:integer</code>      | integer                                        |
| String                 | <code>exi:string</code>       | string, anySimpleType, anyURI, duration, QName |
| n-bit Unsigned Integer |                               | bounded integers                               |
| Unsigned Integer       |                               | Vorzeichenlose INT-Typen                       |
| List                   |                               | alle LIST-Typen, inkl. IDREFS und ENTITIES     |
| QName                  |                               | QName ausschließlich für xsi:type attributes   |

**Tabelle 3.4:** Übersicht aller EXI internen Datentypen

Strings ebenfalls negativ beeinflusst. Im Gegenzug besteht dafür die Möglichkeit, nach Decodierung des EXI Streams wieder das ursprüngliche XML Dokument zu rekonstruieren.

Allerdings ist es gerade die von XML geforderte Flexibilität, die das EXI Format enorm umfangreich und damit auch sehr aufwendig in Hinblick auf eine Implementierung machen. Für die Kommunikation mit Kleinstgeräten ist es meist unerheblich, dass gesamte XML Dokument zu rekonstruieren, da nur gewisse Events erkannt werden müssen, um die entsprechenden Call Back Funktionen aufzurufen.

Eine deutliche Steigerung der Kompaktheit des EXI Streams, bei gleichzeitiger Vereinfachung deren Verarbeitung, stellt die Codierung des XML Dokumentes im Schema informed Modus dar. Hier werden keine Strings mehr, außer die der String Datentypen selbst, übertragen. Die Struktur des XML Dokumentes wird vollständig binär mit Hilfe einer FSM dargestellt. Eine entsprechende Implementierung ist relativ simpel, die Abarbeitung dieser verbraucht nur wenig Arbeitsspeicher, und Rechenleistung wird lediglich für die Decodierung der Daten der Elemente und Attribute benötigt. Damit bietet sich der Schema informed Modus des EXI Formates geradezu für einen Einsatz

auf Kleinstgeräten und Sensorknoten an und sollte in jedem Fall bevorzugt werden. Die Abbildung 3.5 stellt exemplarisch eine Gegenüberstellung beider Codierungsarten anhand des EXI Streams von „notebook.xml“ aus Abbildung 3.4 dar.

```

0x00: 90 01 09 6E 6F 74 65 62 6F 6F 6B 01 01 05 64 61  U..notebook...da
0x10: 74 65 0C 32 30 30 37 2D 30 39 2D 31 32 01 02 01  te.2007-09-12...
0x20: 05 6E 6F 74 65 01 01 09 63 61 74 65 67 6F 72 79  .note...category
0x30: 05 45 58 49 01 01 01 00 01 0C 32 30 30 37 2D 30  .EXI.....2007-0
0x40: 37 2D 32 33 02 02 01 08 73 75 62 6A 65 63 74 03  7-23....subject.
0x50: 01 01 00 01 00 01 05 62 6F 64 79 03 13 44 6F 20  ....body..Do
0x60: 6E 6F 74 20 66 6F 72 67 65 74 20 69 74 21 00 01  not forget it!..
0x70: 01 00 01 00 02 01 00 00 00 00 0F 73 68 6F 70 70  ....shopping
0x80: 69 6E 67 20 6C 69 73 74 00 00 00 0D 6D 69 6C 6B  ing list....milk
0x90: 2C 20 68 6F 6E 65 79 00 01 01                      , honey...

```

„notebook.exi“, non-schema informed, byte-align, 154 Bytes

```

0x00: 90 00 00 00 07 2C 01 00 00 00 05 45 58 49 00 00  U.....EXI..
0x10: 07 F7 00 00 00 00 01 00 00 00 13 44 6F 20 6E 6F  .÷.....Do no
0x20: 74 20 66 6F 72 67 65 74 20 69 74 21 00 00 00 01  t forget it!....
0x30: 00 07 2C 01 00 00 00 0F 73 68 6F 70 70 69 6E 67  ....shopping
0x40: 20 6C 69 73 74 00 00 00 0D 6D 69 6C 6B 2C 20 68  list....milk, h
0x50: 6F 6E 65 79 00 00 01                      oney...

```

„notebook.exi“, schema informed, byte-align, 87 Bytes

**Abbildung 3.5:** „notebook.exi“: non-Schema vs. Schema-informed

# 4 Implementierung des ersten Prototypen

## 4.1 Konzept

Ziel der ersten Implementierung sollte die Programmierung eines Prototypen sein, der in der Lage ist, einen bereits vorhandenen EXI Stream entgegenzunehmen und diesen zu decodieren, um auf bestimmte Events, wie z.Bsp. die Auswertung der in den Elementen und Attributen gespeicherten Daten, zu reagieren. Da für weitergehende Entwicklungsarbeit die Zielplattform TelosB als Sensorknoten vorgesehen ist, wird die Implementierung in ANSI-C vorgenommen. Die Programmierung erfolgte mit Hilfe der Entwicklungsumgebung Qt von Nokia auf einem mit dem Betriebssystem MS Windows 7 betriebenen PC.

Für die Aufgabe des Encodings von XML zu EXI wurde das Open Source Programmbundle „EXIficient“ [1] in der aktuellen Version 0.5 genutzt. Dabei handelt es sich um eine der beiden unabhängigen Implementierungen, die zur Klassifizierung des EXI Formates als Draft-Standards beitragen, und sie wird im großen Umfang durch Siemens Corporate Technology unterstützt. Der Quellcode selbst liegt in JAVA vor. Dem Bundle liegt lediglich das Demo-Programm „EXIficientDemo.java“ bei, welches das Encoding und Decoding einer zuvor spezifizierten XML Datei automatisch vornimmt. Um eine gezielte Codierung vornehmen zu können, wurde dieses Programm in vier verschiedenen Versionen modifiziert:

- XML2EXInoS.java, Konvertierung XML zu EXI, ohne XSD
- XML2EXIS.java, Konvertierung XML zu EXI, mit XSD
- EXI2XMLnoS.java, Konvertierung EXI zu XML, ohne XSD
- EXI2XMLS.java, Konvertierung EXI zu XML, mit XSD

Ohne Angabe weiterer Optionen wird die jeweilige Konvertierung im Byte-aligned Modus durchgeführt, welcher in der Entwicklungsphase sehr hilfreich war. Dieser ließ sich

allerdings erst durch Setzen der entsprechenden Variablen in der Klasse `exiFactory` aktivieren:

```
exiFactory.setCodingMode(CodingMode.BYTE_PACKED);
```

Für diese erste Implementierung wurde auf das XML Dokument „notebook.xml“, welches bereits in Abbildung 3.4 dargestellt ist, inklusive der dazu passenden XSD „notebook.xsd“, zurückgegriffen. Bit- als auch Byte-aligned Modus sollten hierbei unterstützt werden. Die Entwicklungen an einer non-Schema informed Variante wurde im Anfangsstadium bereits eingestellt, da sie für den Einsatz auf dem Zielsystem TelosB für ungeeignet erscheint. Die Verwaltung der dazu nötigen Code-Sätze und Local-Name Tables verbraucht zuviel Speicher und die Kompaktheit dieser EXI Streams ist weit entfernt vom Optimum. Das Hauptaugenmerk liegt nun auf eine Implementierung zur Decodierung eines EXI Streams im Schema informed Modus.

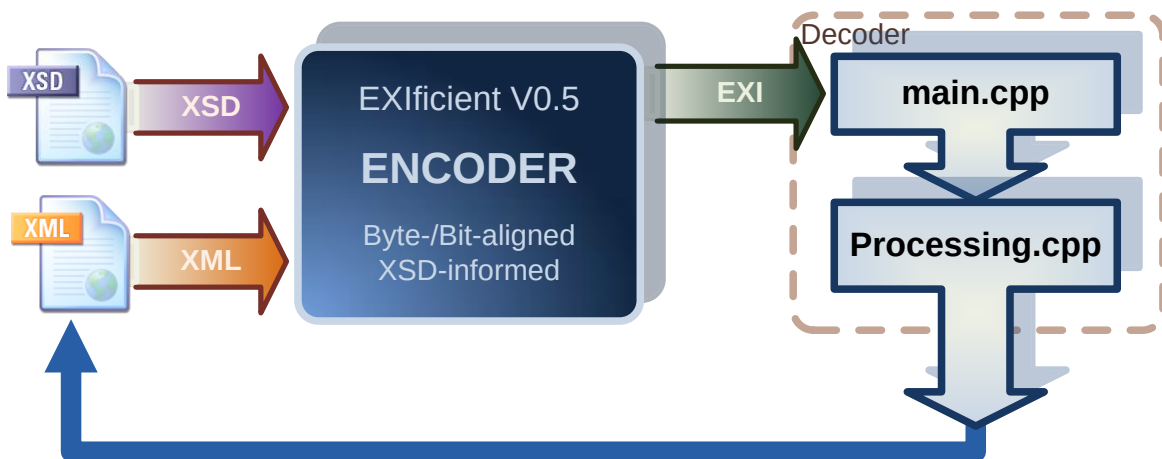


Abbildung 4.1: Funktionsprinzip der Implementierung

Die Applikation, welche als Konsolen-Anwendung ausgeführt wird, besteht gemäß der Abbildung 4.1 aus zwei Teilen:

**main.cpp** Dieser Programmteil stellt die Basis zur Verarbeitung des EXI Streams her. Hier wird die EXI Datei in den Speicher gelesen und evtl. angegebene Übergabeparameter ausgewertet

**processing.cpp** In diesem Programmteil werden alle Funktionen bereitgestellt, die zur Verarbeitung des EXI Streams notwendig sind. Enthalten sind dort unter anderem die Funktion zum Bit-/Byte-weisen Zugriff auf den EXI Stream, Auswertung des Headers, die Verwaltung der String Tables, die Verarbeitung der drei implementierten EXI Datentypen `UInt`, `String` und `dateTime`, sowie die auf der XSD

basierenden FSM.

## 4.2 Programmteil: main.cpp

Dieser Programmteil stellt den Start für die Auswertung des EXI Streams dar. Dazu werden zu Beginn die Übergabeparameter ausgewertet. Auch wenn beliebig viele Übergabeparameter akzeptiert werden, so ist der erste Parameter reserviert für den Dateinamen des zu verarbeitenden EXI Files. Der zweite Parameter ist optional. Als Standard wird für die Verarbeitung des EXI Streams eine Datei im Byte-aligned Modus erwartet. Wurde entsprechendes File jedoch im Bit-aligned Modus codiert, so ist hier die Angabe des Parameters `-b` unbedingt erforderlich, da sonst keine korrekte Verarbeitung möglich ist.

Die Funktion `GetRequest()` öffnet nun daraufhin das entsprechende EXI File im binären read-only Modus und lässt den Dateizeiger durch den Befehl `fseek()` an das Ende der Datei springen. Das Kommando `ftell(file)` liefert nun die Gesamtgröße der Datei in Bytes zurück. Diese Information wird dazu benötigt, um entsprechenden Arbeitsspeicher für die Aufnahme des EXI Streams zu allozieren. Nach Rücksetzen des Dateizeigers an den Anfang der Datei wird der Inhalt des EXI Files byteweise in den zuvor allozierten Puffer eingelesen. Das Ablegen der einzelnen Bytes geschieht dabei unter Verwendung des Datentyps `unsigned char`. Unterschiede bei der Verarbeitung zwischen Byte- und Bit-aligned Stream gibt es in diesem Stadium noch nicht.

Das Hauptprogramm hat damit seine Arbeit erledigt und übergibt die Ablaufkontrolle durch den Aufruf der Funktion `Processing ()` an den zweiten Programmteil.

## 4.3 Programmteil: processing.cpp

Die Funktion `Processing()` ist hier der Einsprungpunkt für die weitere Verarbeitung und Auswertung des EXI Streams. Hierzu werden als erstes durch den Aufruf der Funktion `Init()` die String Tables angelegt. Mit Aufruf der Funktion `Headerproc()` wird der Header des EXI Streams ausgewertet, und im Anschluss wird die FSM gestartet, die den gesamten EXI Stream abarbeitet. Die globale Variable `bool Bitalign`, welche durch die Funktion `void setAlignment(bool align)` aus der `main.cpp` gesetzt wird, spezifiziert den vorliegenden Stream als Byte- oder Bit-align gemäß den Übergabeparametern. Im Folgendem seien die wichtigsten Programmteile zur Verarbeitung des EXI Streams hier kurz erwähnt.

**Init()** In dieser Funktion werden die String Tables angelegt. Da eine strikte Abarbeitung des EXI Streams gemäß der XSD vorgesehen ist, entfällt hier das Anlegen der Local-Names Table. Die URI Table wird gemäß der Abbildung 4.2 als zweidimensionales Array mittels dynamischer Speicherverwaltung angelegt und für die Verarbeitung von EXI Streams, die mit Hilfe von XML Schema erzeugt wurden, mit den vier Standardeinträgen gem. der Abbildung 4.2 initialisiert. Die Prefix Table und die Value Table werden ebenso, allerdings ohne Einträge, deklariert und existieren in der Startphase als reine Zeiger in den Arbeitsspeicher. Für die Speicherung aller Strings wird ein dynamisch reserviertes eindimensionales Array vom Datentyp `unsigned char` vorgesehen, um nur soviel Speicher wie eben nötig zu verbrauchen. Auch wurde auf die Verwendung von C Strings zugunsten der Verwendung sogenannter Pascal String verzichtet. Das erste Byte eines Strings gibt damit immer die Länge des tatsächlichen Strings an. Entgegen den C Strings, die erst den gesamten String nach dem Sonderzeichen NULL durchsuchen müssen, steht damit von vornherein fest, wie lang der gespeicherte String tatsächlich ist. Vergleichsoperation können damit sofort abgebrochen werden, sobald sich die Länge der Strings unterscheidet.

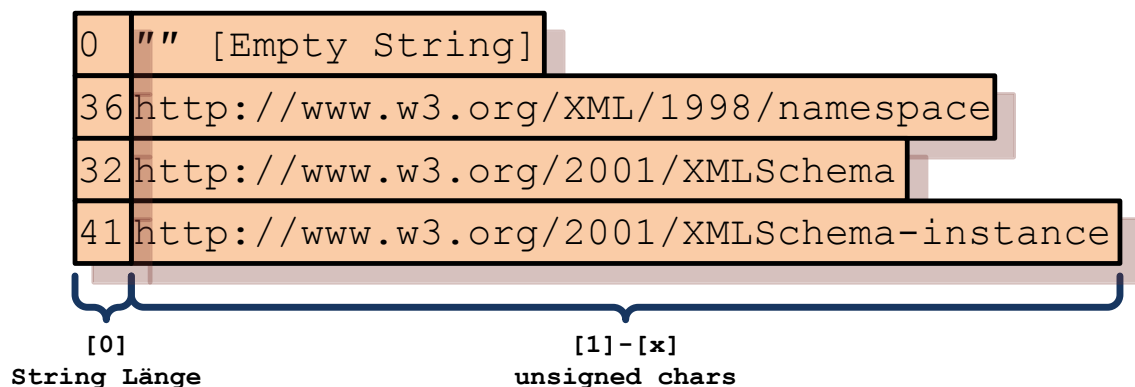


Abbildung 4.2: URI Table im Initialisierungszustand

**Headerproc()** Hier erfolgt die Auswertung des Headers, welcher im Puffer des EXI Streams gespeichert wurde. Dazu wird als erstes nach der Zeichenkette `$EXI` gesucht, und sie wird bei Vorhandensein in der Konsole ausgegeben. Da sie ein rein optionales Element im Header darstellt, wird bei nicht Vorhandensein dieser mit der Auswertung der Distinguishing Bits fortgefahren. Ein nicht Vorhandensein dieser bricht die weitere Verarbeitung des EXI Streams allerdings sofort ab, da sie zwingend vorausgesetzt werden. Anschließend wird das Presence Bit for EXI Options eingelesen, welches aber in der weiteren Verarbeitung nicht mit einfließt, da EXIficient zur Zeit keine Angabe von Optionen unterstützt. Es erfolgt noch eine Auswertung der Versionsnummer gemäß der Tabelle 3.2. Auch diese Infor-

mation bleibt ohne Konsequenz für die weitere Verarbeitung des vorliegenden EXI Streams. Durch EXIficient V0.5 wird eigentlich immer nur der Standardheader mit der Größe von einem Byte erstellt, der den Wert 0x90 enthält. Die hier getätigten Auswertungen des Headers dienen eher zukünftigen Entwicklungsarbeiten.

**getbit()** Mittels dieser Funktion erfolgt der Zugriff auf den im Puffer gespeicherten EXI Stream. Als Übergabeparameter wird dort die Adresse eines `unsigned char` (Call by Reference), sowie die Anzahl der auszulesenden Bits angegeben. Da dieser Datentyp nur ein einziges Byte umfasst, können damit max. 8 Bit ausgelesen werden. Zwischen Byte- und Bit-aligned Modus wird hier nun durch die Abfrage der globalen Variablen `Bitalign` unterschieden. Bei der Anforderung im Byte-aligned Modus mit der Anzahl der auszulesenden Bits größer Null, wird dabei immer das gesamte Byte aus dem EXI Stream zurückgeliefert. Im Bit-aligned Modus wird ein Byte auf die angegebene Adresse geschrieben, welches durch entsprechende Schiebeoperationen den Wert ab der Position des LSB enthält. Nicht angeforderte Bits werden zum MSB hin mit NULLEN gepaddet. Um die Position innerhalb des EXI Streams eindeutig bestimmen zu können, existieren die zwei globalen Variablen `incnt`, welche auf das aktuelle Byte zeigt, und `inbitcnt`, welche auf das aktuelle Bit innerhalb des aktuellen Bytes zeigt und damit den maximalen Wert `inbitcnt = 7` annehmen kann. Da eine Funktion zur Berechnung des `ld`, welche für die Anforderung der entsprechenden Anzahl von Bits notwendig ist, standardmäßig in keiner C Library enthalten ist, wurde hier die Hilfsfunktion `ld()` implementiert. Sie basiert auf der Formel 4.1 zur Berechnung der notwendigen Bitlänge  $l$ .

$$l = \left\lceil \frac{\log_{10} m}{\log_{10} 2} \right\rceil \quad (\text{m: Anzahl/Wertigkeit}) \quad (4.1)$$

**getDATETIME()** Durch Aufruf dieser Funktion werden die Daten des in EXI eingebauten Datentyps `dateTime`, welcher in der XSD Spezifikation mit gleichem Namen existiert, decodiert und ausgewertet. In der Beispiel XML Quelldatei kommt dieser in Verbindung der Attribute `date` vor und wird nach Auftreten des Events `AT(date)` ausgewertet. Dabei wird auf eine Datenstruktur, welche in der zugehörigen Header Datei `processing.h` definiert ist, zurückgegriffen. Sie basiert auf einem `struct` namens `date` und enthält die Werte für die Jahreszahl, den Monat und den Tag. Die Jahreszahl wird zu Beginn mit dem Wert 2000, eine zweite Variable namens `monthday` mit 0 initialisiert. Danach wird aus dem EXI Stream der Jahreszahl-Offset, welcher aus dem ein Bit langen Vorzeichen

und einem `unsigned Integer` Wert besteht, ausgelesen und, je nach Vorzeichen, vom Initialisierungswert addiert oder subtrahiert. Nach Bestimmung der Jahreszahl wird der Variablen `monthday` durch Einlesen der nächsten 9 Bit aus dem EXI Stream als `unsigned Integer` ihr Wert zugewiesen. Der tatsächliche Monat wird dann durch die Operation  $monthday/32$  ermittelt und gespeichert. Die Tageszahl ergibt sich mit Hilfe der Modulo-Operation  $monthday\%32$ . Das gesamte Datum wird dann noch innerhalb der Funktion auf der Konsole ausgegeben. Für die Bildung der Variable `monthday` zeichnet sich die Formel 4.2 verantwortlich. Danach wird noch ein einzelnes Bit zur Präsenz der Zeitzoneangabe eingelesen, welches aber in der vorliegenden Implementierung nicht ausgewertet wird. Die Angabe einer Zeitzone ist hier rein optional und das zuletzt eingelesene Bit wird mit der Wertigkeit 0, es existiert keine Angabe zur Zeitzone, vorausgesetzt.

$$monthday = Monat * 32 + Tag \quad (4.2)$$

**getSTRING()** Nach Eintreten eines entsprechenden Events vom Typ Attribut **AT** oder Character **CH** auf einen String Datentypen wird diese Funktion aktiviert und zeigt im EXI Stream auf einen `unsigned Integer` Wert. Dieser wird mittels der Funktion `getUint()` ausgewertet und gibt drei Möglichkeiten vor, den auftauchenden String zu klassifizieren. Dazu wurde in der Funktion `Init()` bereits die entsprechende Value Table als dynamisch verwaltetes zweidimensionales Array vom Datentyp `unsigned char` angelegt. Die erste Spalte, angesprochen über die Variable `Value_Table[y][0]`, gibt die absolute Länge des Strings wieder. Die zweite Spalte, namentlich als `Value_Table[y][1]` bezeichnet, enthält eine State Nummer als ID. Erst danach folgt der String, wie in der Abbildung 4.3 zu erkennen ist. Der Aufruf der Funktion erfolgt immer mit der State Nummer als Übergabeparameter. Dieser wird hier bei Auslesen einer 0 als `unsigned Integer` als Owner ID für das Auftreten eines bereits gespeicherten Strings via Local ID interpretiert. Die Länge der nun einzulesenden Bits ergibt sich aus der Anzahl aller unter dieser Owner ID gespeicherten Werte in der Value Table unter Anwendung der Formel 4.1. Der entsprechende String wird gemäß seiner Auftrittsreihenfolge anhand dieser eingelesenen Bitfolge im Value Table lokalisiert. Steht der Zeiger nach Auftreten der entsprechenden Events dagegen auf einer 1, so wird hier auf einen bereits gespeicherten String mittels Global ID verwiesen. Die nun einzulesende Bitlänge bezieht sich auf die aktuelle Gesamtzahl aller Einträge in der Value Table und verweist mit ihrer Wertigkeit direkt auf den Index der Value Table. Tritt dagegen nach dem Event ein Wert größer gleich zwei auf, so handelt es sich hierbei um einen neuen, nicht gespeicherten String. In diesem Falle wird eine neue

Zeile innerhalb der Value Table angelegt und der String mit der Länge minus zwei des aufgetretenen Wertes vom Puffer in die Value Table übertragen. Als Owner ID wird hier der Übergabeparameter zum Aufruf der Funktion gespeichert und der globale Zähler `entriesValues` um eins erhöht, der die aktuelle Gesamtzahl der gespeicherten Strings in der Value Table angibt. Als Rückgabewert liefert die Funktion `getSTRING()` eine Slot ID, welcher der globalen ID und damit dem Zeilen Index der Value Table entspricht. Mit der Hilfsfunktion `printstring()` kann damit der entsprechende String gemäß seiner Slot ID aus dem Value Table auf der Konsole ausgegeben werden.

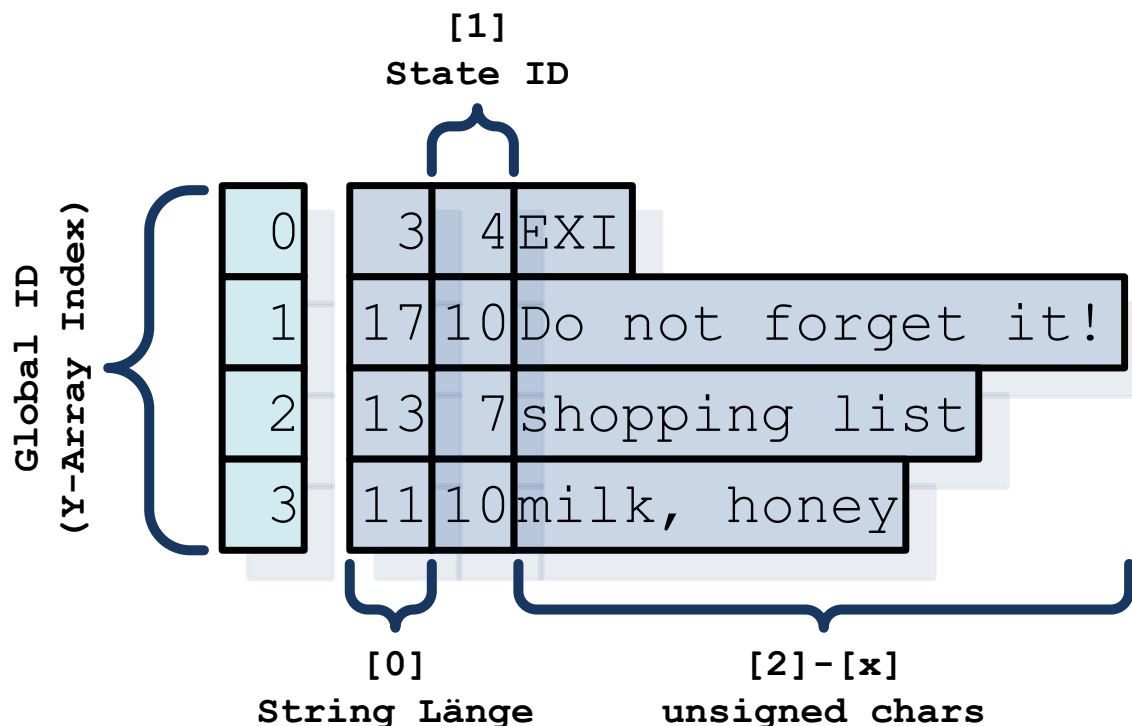


Abbildung 4.3: Vollständige Value Table von „notebook.exi“

**getUint()** Diese Funktion ist elementar notwendig zur Verarbeitung von String Values und gibt nach Aufruf einen `unsigned Integer` Wert zurück. Dazu wird ein Multiplikator, hier die Variable `mult`, mit einer 1 und ein Rückgabewert, hier `value`, mit 0 initialisiert. Anschließend wird ein volles Oktet, also 8 Bit in Folge, eingelesen und deren MSB ausgewertet. Ist dieses 0, so besteht der Wert des `unsigned Integer` Wertes aus den verbleibenden sieben Bits. Steht dagegen an der Stelle des MSB eine 1, so ergibt sich der Rückgabewert `value` aus der Aufsummierung von den eingelesenen 8-Bit (7-Bit ohne MSB) multipliziert mit dem Multiplikator. Letzter wird dann mit 128 multipliziert und das nächste Oktet wird eingelesen. Ist auch das MSB von diesem eine 1, so wiederholt sich der geschilderte Vorgang.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">

  <xs:element name="notebook">
    <xs:complexType>
      <xs:sequence maxOccurs="unbounded">
        <xs:element name="note" type="Note"/>
      </xs:sequence>
      <xs:attribute ref="date"/>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="Note">
    <xs:sequence>
      <xs:element name="subject" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>

    <xs:attribute ref="date" use="required"/>
    <xs:attribute name="category" type="xs:string"/>
  </xs:complexType>

  <xs:attribute name="date" type="xs:date"/>
</xs:schema>
```

Abbildung 4.4: XML Schema „notebook.xsd“

**FSM.notebook()** Das Herz der gesamten Verarbeitung des im Schema informed Modus codierten EXI Streams ist die Zustandsmaschine. Diese wurde auf Grundlage des vorliegenden XML Schemas gemäß der Abbildung 4.4 durch Auswertung per „Hand“ erstellt. Hier erfolgt nur die Erkennung der Event Codes und der Sprung zu den Call-Back Funktionen, welche die Auswertung der relevanten Daten übernehmen. Die FSM wird als letzter Verarbeitungsschritt innerhalb der Processing() gestartet und besitzt als einzigen Übergabeparameter eine State ID, die hier natürlich auf den Ausgangszustand 0 zeigt. Entsprechend den Vorgaben aus der XSD wird nun bei einem State Wechsel die dazu nötige Bitlänge ausgelesen, welche zum Verzweigen zum nächsten State führt. Die FSM ist dabei als Rekursive Funktion definiert, welche als Übergabeparameter den nächsten State

```
Using inputfile for request: notebook_byte.exi
Create outputfile for reply: notebook.out
Inputfilesize / Bytes: 87
Final Preview Version 1.0 detected !
State: 0 --> Start Dokument
State: 1 --> StartTag notebook
State: 2 --> Attribut: date
Year: 2007
Month: 9
Day: 12
State: 3 --> StartTag note
State: 4 --> Attribut: category
category: EXI
State: 5 --> Attribut: date
Year: 2007
Month: 7
Day: 23
State: 6 --> StartTag subject
State: 7 --> Value subject
EXI
State: 8 --> End Element subject
State: 9 --> StartTag body
State: 10 --> Value body
Do not forget it!
State: 11 --> End Element body
State: 12 --> End Element note
State: 3 --> StartTag note
State: 5 --> Attribut: date
Year: 2007
Month: 9
Day: 12
State: 6 --> StartTag subject
State: 7 --> Value subject
shopping list
State: 8 --> End Element subject
State: 9 --> StartTag body
State: 10 --> Value body
milk, honey
State: 11 --> End Element body
State: 12 --> End Element note
State: 13 --> End Element notebook
State: 14 --> End Dokument
```

**Abbildung 4.5:** Konsolenausgabe nach Decodierung von „notebook.exi“

enthält. Somit wird gewährleistet, dass die FSM sich vollständig selbstständig, entsprechend den Definitionen des XML Schemas, abarbeitet, bis der gesamte EXI Stream verarbeitet ist. Treten die Events Attribut AT oder Character CH auf, so erfolgt ein Einsprung in Funktionen, die die entsprechenden Daten verarbeiten. Auch hier ist es notwendig, dass der gesamte Bitstream in der richtigen Reihenfolge vollständig abgearbeitet wird. Abbildung 4.6 zeigt die gesamte FSM mit allen 14 States. Der in der Farbe rot dargestellte State ist dabei der Einsprungspunkt, der mit gelb Gefüllte, mit braunen Rand versehene State, stellt das Ende der FSM dar. Die Notation der State-Übergänge besteht hier aus dem Eventcode als erste dezimale Ziffer. Die folgende tiefgestellte Ziffer gibt die Bitlänge des Eventcodes an. Die Bezeichnung der einzelnen States entspricht der in Tabelle 3.3 aufgeführten Events. Die in jedem State in letzter Zeile stehende Ziffer stellt die State ID als laufende Nummer dar. Was aus dieser Abbildung nicht hervorgeht: jeder State besitzt hier noch eine weitere Verzweigungsmöglichkeit.

Dies ist die Verzweigung zum State **Undeclared**, da die gesamte Codierung im non-strict Modus erfolgt. Bei Auftreten eines nicht definierten State-Wechsels wird dies registriert und führt zum kontrolliertem Programmabbruch. Die Anzahl der für den State Wechsel einzulesenden Bits richtet sich auch hier nach den Verzweigungsmöglichkeiten+1 betrachtet als  $m$  nach Anwendung der Formel 4.1. Nach Erreichen des FSM-Endzustandes **ED** kehrt die Programmablaufkontrolle zur **Processing()** zurück, die wiederum den Rücksprung in die **main.cpp** vollzieht, wodurch die Auswertung des EXI Streams als erledigt betrachtet und das Programm beendet wird.

Erst durch Verwendung der Codierung von XML Dokumenten, die sich strikt an der entsprechenden XSD halten, ist eine ressourcensparende Implementierung zur Auswertung des EXI Streams möglich. Durch die Implementierung einer rekursiven FSM, die als einzigen Übergabeparameter die State ID benötigt, existiert die Möglichkeit, variable XML Dokumente basierend auf einer einzigen XSD auch auf Kleinstgeräten zu verarbeiten. Entsprechend der Verarbeitung innerhalb der FSM kann auch hier das ursprüngliche XML Dokument, wie in Abbildung 4.5 zu sehen ist, rekonstruiert werden.

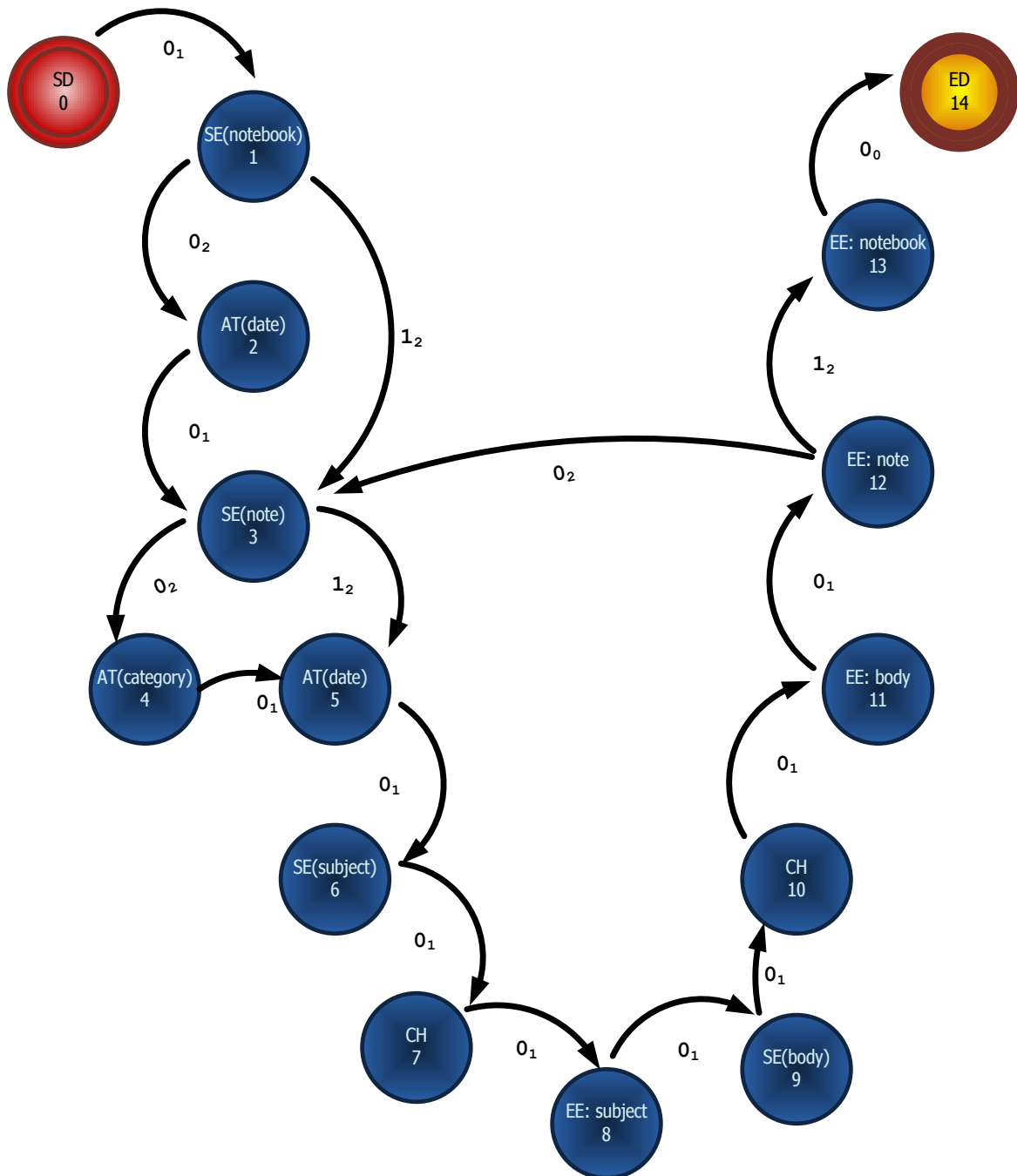


Abbildung 4.6: Vollständige FSM zu „notebook.xsd“

# 5 Implementierung des Codegenerators

## 5.1 Motivation

Wie in dem vorangegangenen Kapitel ersichtlich wird, entstehen selbst durch eher kleine XSDs bereits umfangreiche FSMs. Komplexere XSDs bzw. solche, die sich aus mehreren XML Schemas zusammensetzen, mit der „Hand“ zu analysieren, um dadurch die entsprechenden FSMs zu kreieren, stellt sich dabei als äußerst mühseliges und fehleranfälliges Vorgehen dar. Als sehr unpraktisch erweist sich auch die Tatsache, dass bei Änderung der verwendeten XSD auch die Implementierung der FSM im Quelltext geändert werden muss. Ziel im weiteren Schaffen dieser Projektarbeit sollte es damit sein, die Erstellung der FSM auf Grundlage beliebiger XSDs zu automatisieren und den C Quelltext generieren zu lassen.

Im ersten Schritt wird dazu die FSM in der vorliegenden Implementierung vollständig aus der `processing.cpp` entfernt. Diese soll nach Auswertung des XML Schemas als eigenständige Quelltextdatei dem Projekt hinzugefügt werden. Damit sind die Quelltexte der `processing.cpp` und `main.cpp` absolut statisch und damit unabhängig von der verwendeten XSD.

Um an die Daten zu gelangen, die der entsprechenden XSD entstammen und mit deren Hilfe die FSM erzeugt werden soll, wird die Implementierung von EXIficient V.0.5, deren JAVA Quellcode als Open Source Project zugänglich ist und vorliegt, etwas näher durchleuchtet werden. In der entsprechenden Klasse dieses Projektes sind die Daten zu analysieren, zu extrahieren und für das Erzeugen einer entsprechenden FSM, ähnlich der ersten Implementierung, aufzubereiten. Mittels der modifizierten und hinzugefügten Methoden in EXIficient selbst, kann dann die FSM in einer separaten `cpp` Datei mittels Stringverarbeitung ausgegeben werden. Die so erzeugte C Quelldatei wird dann im letzten Schritt in das Gesamtprojekt eingebunden.

## 5.2 Implementierung

Alle zur Generierung der FSM notwendigen Daten wurden innerhalb des EXIficient Projectes innerhalb der Klasse **XSDGrammarBuilder** lokalisiert. Innerhalb der Methode **toGrammar()** erfolgt der Zugriff auf sie, so dass hier die Modifikationen beginnen.

Im ersten Schritt werden dazu innerhalb dieser Methode zwei Dateien, gelegen im selben Verzeichnis wie die XSD selbst, angelegt: die **cpp**-Datei, welche die zukünftige FSM beinhaltet, und ein zugehöriges Header File mit der Dateiendung **.h**. Den Dateinamen erhält die Methode aus der Klassenvariablen **FSM\_File**, die nach Aufruf der zusätzlichen Methode **getDstFile()** aus der Klasse **GrammarFactory** heraus befüllt wird. In diesen beiden Quelldateien werden anfangs die notwendigen Initialisierungseinträge, wie z. Bsp. **includes**, geschrieben. Reinen Informationscharakter haben dagegen das Schreiben der Kommentare mit Angabe, ob der strict-mode verwendet wurde und die vollständige URI Table in die **cpp** Datei. Die URI Table liegt bereits als finale Datenstruktur, namentlich **uris**, in Form eines Stringarrays vor und wird vollständig mit vorangestelltem Arrayindex geschrieben. Dies erfolgt allerdings nur, sofern mehr als nur die vier standardmäßigen URIs in ihr vorhanden sind.

Entgegen dem Konzept der ersten Implementierung, liegt bei EXIficient das codierte XML Schema nicht als ein vollständiger Graph vor. Hier wird jedes mögliche Element als eigenes Fragment in der Datenstruktur **fragmentElements** alphabetisch sortiert nach seinem **qname** gespeichert. Abbildung 5.1 zeigt einen Ausschnitt dieser Container Datenstruktur. Im Fall der **notebook.xsd** sind hier also die Fragmente **body**, **note**, **notebook** und **subject** enthalten. Um aus den in den Fragmenten enthaltenen Daten die FSM zu generieren, wird eine eigene Datenstruktur, hier als Klasse **State**, angelegt. Sie dient der Aufnahme aller relevanten Daten und besteht aus den folgenden Elementen:

- **String Event**, Eventtyp in Klartext
- **String Data**, Datentyp bei CH, bei AT() zusätzlich Attributnamen, bei SE() **qname**
- **int Slots**, Anzahl der Weiterverzweigungen
- **int[] Kanten**, Ziel State ID

Die States selbst werden als Array organisiert und über den laufenden Feldindex angesprochen. Jeder State stellt dabei einen Knoten innerhalb der FSM dar und wird durch seine Eventbezeichnung und den evtl. vorhandenen Datentyp ausreichend klassifiziert. Das Integer Array **Kanten** enthält die Verzweigungen zu nachfolgenden States.

Der Feldindex selbst ist dabei der zu verwendete EXI Eventcode, der Inhalt im Feld zeigt auf den Index des States.



Abbildung 5.1: Ausschnitt aus der Datenstruktur zu `fragmentElements`

Nun folgt die Abarbeitung eines jeden Fragmentes. In der ersten Ebene eines Elementes steht dabei immer ein Start Element SE, dieser wird automatisch immer zum `State[0]` einer jeden FSM. Der `qname`, der dort unter `State[0].Data` abgelegt wurde, wird gleichfalls in der Klassenvariablen `aktQname` gespeichert, womit die FSM ihren Namen erhält. Erst jetzt sind weitere Verzweigungen möglich. Das weitere Durchhangeln durch den Graphen des Fragmentes übernimmt an dieser Stelle die zusätzliche Methode `StateListing()`. Diese ruft sich solange rekursiv nach dem Prinzip der Tiefensuche selbst auf, bis alle Knoten des Graphen besucht wurden. Dabei werden die Daten des Stammknoten in den entsprechenden `State[x]` unter `Event` und `Data` gespeichert und die nächste Verzweigung eine Ebene tiefer untersucht. Wird hier ein bekannter State angetroffen, so wird deren State ID in das Array der Kanten des Stammstates gespeichert. Ein noch unbekannter State wird als Neuer initialisiert und die neu vergebene State ID in das Array der Kanten gespeichert. Ein rekursiver Methodenaufruf endet an der Stelle, deren aktueller Eventtyp ein `EndElement (EE)` ohne weitere Verzweigung ist. Nun wird der nächste auf dem Befehlsstack liegende Methodenaufruf abgearbeitet. Die Zusatzmethode `addSlot()` übernimmt dabei jedesmal die Überprüfung, ob ein State bereits im Statearray eingetragen ist. Auch das Anlegen neuer States geschieht hier. Da es sich hier um einen Graphen handelt, können auch Zyklen auftreten, durch die die Tiefensuche in einer Endlosschleife gerät. Diesen gilt es zu detektieren und zu

vermeiden. In dieser Methode geschieht das durch die Überprüfung, ob die Kante vom Start- zum Zielknoten bereits existiert. Bei Vorhandensein dieser, wird der Methodenaufruf ebenfalls beendet.

Wurden alle Knoten eines Fragmentes besucht, so wird die hinzugefügte Methode `MakeMachine()` aufgerufen. Durch sie erfolgt nun das Generieren des C Quellcodes für eine FSM auf Grundlage der gesammelten Daten im Array der States. Dabei wird aus jedem State eine Case-Verzweigung in der obersten Switch-Anweisung der FSM. Bei der Auswertung eines jedes States wird zwischen den Eventtypen `ATTRIBUTE`, `CHARACTERS` und `START_ELEMENT` unterschieden. Handelt es sich bei dem zu verarbeitendem State um ein Attribut, so wird der Aufruf einer Call-Back Funktion generiert. Dieser setzt sich aus „getAttrib“, dem Namen der FSM und dem Attributnamen zusammen. Handelt es sich beim vorliegenden State um das Datenevent CH, wird ebenfalls ein Call-Back Funktionsaufruf, in der Form „getValue“ mit Namen der FSM zuzüglich des EXI Datentyps, generiert. Die Namen der Call-Back Funktionen, sowie die verwendeten Datentypen werden dabei während der Verarbeitung fortlaufend in der Klasse `ValueHelper` zur späteren Verarbeitung gespeichert. Tritt ein Start Element auf, dessen State nicht die `ID = 0` besitzt, so handelt es sich um einen Sprung von der aktuellen FSM hinein in eine andere. Damit erreicht man die ursprüngliche Verschachtelung der einzelnen Fragmente. Hier wird der Funktionsname der FSM bestehend aus „FSM“ zzgl. dem `qname` der Ziel-FSM erzeugt. Als Startstate wird stets eine 0 als Übergabeparameter mit angegeben. Anschließend wird der Befehl zum Einlesen der entsprechenden Bits generiert. Die Anzahl ergibt sich dabei aus der Anzahl der Verzweigungsmöglichkeiten (Slots) zzgl. dem `Undeclared` State. Sofern nötig, wird nun eine switch-Anweisung generiert, die den nächsten State bestimmt. Sofern dieser existiert, folgt zum Schluss der rekursive Aufruf für die eigene FSM, der als Übergabeparameter die neue State ID enthält. Ist die FSM eines Fragmentes vollständig geschrieben, so wird die aktuelle Instanz der Klasse `State` gelöscht und eine neue initialisiert und das nächste Fragment abgearbeitet, sofern vorhanden.

Nun erfolgt das Generieren aller Call-Back Funktionen auf Grundlage der in der Klasse `ValueHelper` gesammelten Daten. Die Funktionen enthalten dabei stets einen Funktionsaufruf zum Auslesen der Daten aus dem EXI String. Für die bereits in der `processing.cpp` implementierten Datentypen `String` und `dateTime`, werden die Funktionsaufrufe bereits syntaktisch korrekt generiert. Für alle anderen Datentypen wird ein generischer Aufruf verwendet. Die Funktionsnamen selbst werden ebenso auch im zugehörigen Header File gespeichert. Sind in der URI Table mehr als nur die standardmäßig vorhandenen vier Einträge gespeichert, so enthält jeder Funktionsname der Call-Back Funktionen noch eine ganze Zahl, die als Arrayindex auf die entsprechende

URI hinweist. So wird sichergestellt, dass mehrere Funktionen nicht mit dem selben Namen erzeugt werden, sondern hinsichtlich ihres Namespaces bereits auseinander gehalten werden können. Die Auswertung und Zuweisung eines `qnames` zu einem Namensraum übernimmt dabei die Zusatzmethode `getName()`. Abschließend wird im `cpp`

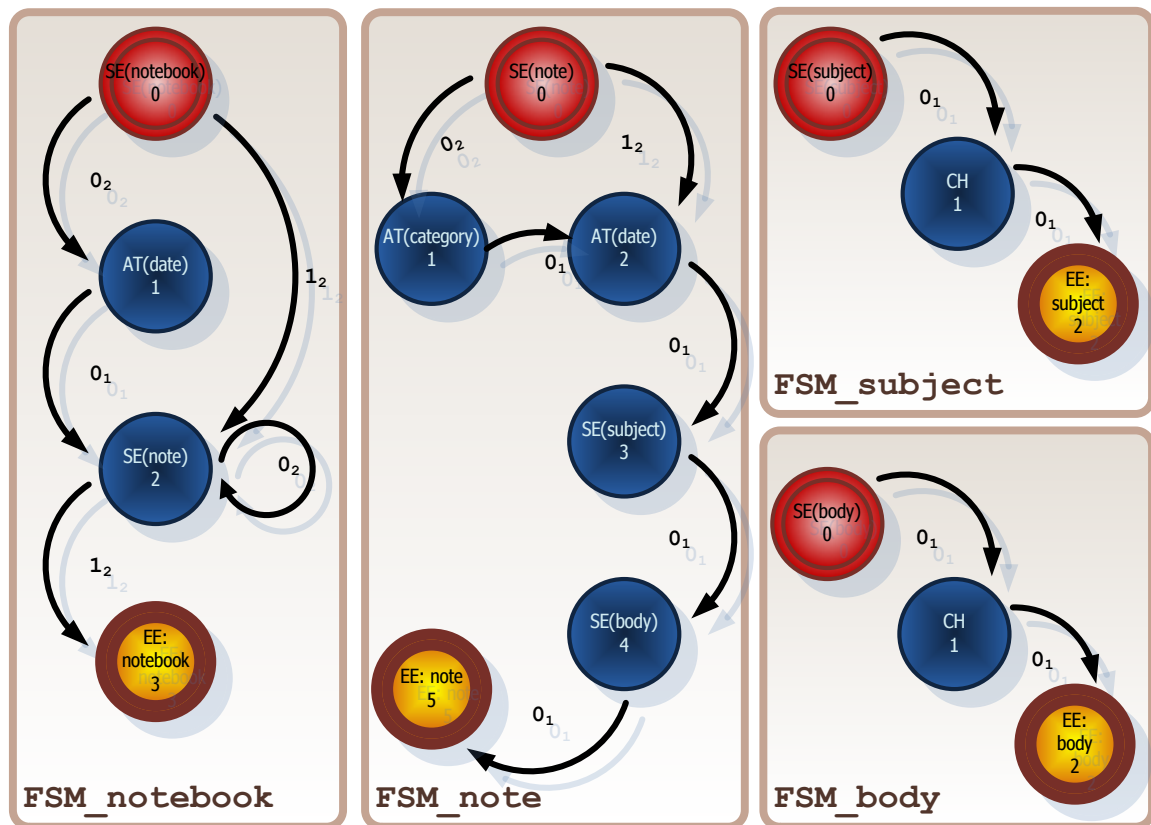


Abbildung 5.2: notebook.xsd als fragmentierte FSM

File noch die Funktion `FSM_Start()` generiert. Sie enthält die Anweisung zum Start der FSM, welche das Root-Element darstellt. Die Information dazu wird aus der EXIficient Datenstruktur `globalElements` bezogen. Hier sind nochmals alle Fragmente enthalten, die als Root-Element auftreten können. Im Fall der `notebook.xsd` existiert dort nur ein einziger Eintrag, der das Fragment `notebook` als Root-Element kennzeichnet. Sind dort jedoch mehrere Fragmente gelistet, so werden Anweisungen im Quellcode generiert, die als erstes den DocContent Code mit entsprechender Anzahl an Bits aus dem EXI Stream auslesen. Anschließend wird eine switch-Verzweigung erzeugt, die entsprechend der Codierung die entsprechende FSM startet.

Nach Schließen der beiden Quelldateien liegt nun eine vollständig automatisch generierte Implementierung einer FSM vor, die im einfachsten Fall nur noch dem Projekt hinzugefügt werden muss. Die Abbildung 5.2 zeigt exemplarisch die aus der `notebook.xsd` hervorgegangenen vier FSMs als Fragmente.

## 6 Komplexität und Validierung der Implementierung

Die Entwicklungsarbeiten der ersten Implementierung basierten ausschließlich auf die Umsetzung des XML Schemas `notebook.xsd` als statische FSM, damit das Standardbeispiel `notebook.xml` vollständig und fehlerfrei decodiert werden konnte. Das Ziel bestand hier nicht nur aus dem Analysieren der zugehörigen XSD und den Aufbau der korrekten FSM zur Detektion der Eventcodes, sondern auch um den ersten Test der Hilfsfunktionen, wie z.B. `getbit()`, und den datenverarbeitenden Funktionen für die drei Datentypen `dateTime`, `String` und `Unsigned Integer`. Dabei wurde anfangs mit Byte-aligned codierten EXI Streams gearbeitet, da sich das Nachvollziehen von Fehlern anhand dieser Streams deutlich vereinfacht. Da die Verarbeitung von Bit-aligned Streams vorzuziehen ist, wurde die Unterstützung beider Moden von vornherein vorgesehen und in der Implementierung berücksichtigt. Nach den ersten erfolgreichen Durchläufen, ganz gleich ob Bit-oder Byte-aligned, wurde das originale XML Dokument in zwei weiteren Varianten modifiziert und erweitert, die aber dennoch valid zum vorgegebenem XML Schema sind. Die selbsterstellte FSM und die restlichen Funktionen konnten dabei ihre Funktionalität und korrekte Abarbeitung beweisen, indem alle Streams zuverlässig decodiert wurden.

Bei der Implementierung des C Codegenerators in JAVA war es vom großen Vorteil, dass durch das Vorhandensein der XSD Grammatik als Datenstruktur der Soll-Zustand der zu erzeugenden FSM bereits vorgegeben ist. Von diesem Punkt an war es möglich, beliebige andere, teils deutlich komplexere XML Schemas als FSM erzeugen zu lassen. Dabei wurden weitere FSMs zu diversen anderen einfach gehaltenen XSDs erstellt und vollständig mit der in EXIficient vorliegenden Datenstruktur verglichen. Die deutlich komplexeren XML Schemas zu den Webservices WSDL und SOAP, konnten dabei ebenfalls als FSM erstellt werden. Durch ihre deutlich höhere Komplexität erfolgte hier ein Abgleich zur Datenstruktur aber nur stichprobenartig.

Dadurch, dass nur die oben genannten drei Datentypen unterstützt werden, und derzeit noch keine Verarbeitung von Namespace Events, die innerhalb der EXI Streams

auftreten können, in der vorliegenden Implementierung durchgeführt wird, beschränkte sich das Testen zur Decodierung auf einige wenige, auf Stringdaten basierende EXI Streams. Diese konnten aber fehlerfrei im Bit- als auch im Byte-aligned Mode decodiert und die in ihnen enthaltenen Daten verarbeitet werden.

Bei der Decodierung von EXI Streams, deren ursprüngliche XML Dokumente sich strikt an deren XSD halten, sind die Anforderungen an den Arbeitsspeicher und der Rechenkapazität durch Verwendung der in dieser Implementierung vorliegenden FSM sehr gering. Jeder State-Wechsel innerhalb einer FSM verbraucht dabei lediglich den Speicher einer **Integer**-Variable, welche die aktuelle State ID symbolisiert und einen Platz auf dem Befehlsstack durch den rekursiven Aufruf der FSM. Dabei stellt die Verwendung der fragmentierten FSMs einen großen Vorteil hinsichtlich des Speicherbedarfs dar, da abgearbeitete Fragmente der unteren Ebenen wieder vollständig vom Stack und dem Arbeitsspeicher entfernt werden. Durch die konsequente Nutzung der dynamischen Speicherverwaltung für die String Tables wird ebenfalls nur soviel Arbeitsspeicher in Anspruch genommen, wie es eben notwendig ist.

## 7 Zusammenfassung und Ausblick

Das EXI Format stellt ein äußerst mächtiges und sehr flexibles Werkzeug zur vollständigen binären Repräsentation von XML Dokumenten dar. Dessen Implementierung zur Gewährleistung des vollen Funktionsumfanges zur Codierung der EXI Streams fällt jedoch ähnlich komplex aus, wie die Verarbeitung der XML Dokumente selbst. Zwar verzichtet das EXI Format vollständig auf den Einsatz eines speicherintensiven Parsers zur Verarbeitung des Streams, aber der nach wie vor bestehenden hohen Flexibilität ist ein hoher Tribut in Sachen Rechenleistung und Implementierung zu zahlen. Die Vorteile und damit eine zukünftig zunehmende Akzeptanz bei Verarbeitung dieser auf Kleinstgeräten, spielt das EXI Format auf die Spezialisierung bestimmter und besonders effektiver Teilbereiche, die dieses Format bietet, aus. Eine Übersicht, über die anhand der Beispiel-Datei `notebook.xml` sich ergebenden Speicherplatzeinsparungen durch die Verwendung des EXI Formates sind der Tabelle 7.1 zu entnehmen.

Für die Verarbeitung auf ressourcenlimitierten Plattformen sind besonders EXI Streams geeignet, welche im Schema informed Modus codiert wurden. Durch Verwendung dieser, wird das zu codierende XML Dokument durch ein enges Modell definiert, dessen Auswertung anhand einer FSM vonstatten geht. Dabei wird zum Einen Speicherplatz und Rechenkapazität durch Abarbeitung derer eingespart, auch der Umfang der nötigen Implementierungsarbeiten hält sich in enge Grenzen. Die dazu erfolgte Implementierung eines Code-Generators zur Erzeugung der entsprechenden FSM erleichtert die Verarbeitung derartiger EXI Streams.

| Codierung XML -> EXI                    | Dateigröße in Bytes | Größenverhältnis |
|-----------------------------------------|---------------------|------------------|
| ohne, reines XML Dokument               | 321                 | 100 %            |
| Schemalos, Byte-Align                   | 154                 | 48 %             |
| Schemalos, Bit-Align                    | 124                 | 39 %             |
| Schemabehaftet, Byte-Align              | 87                  | 27 %             |
| Schemabehaftet, Byte-Align, strict-mode | 75                  | 23 %             |
| Schemabehaftet, Bit-Align               | 61                  | 19 %             |
| Schemabehaftet, Bit-Align, strict-mode  | 59                  | 18 %             |

**Tabelle 7.1:** Größenvergleich zur Codierung der `notebook.xml` im EXI Format

Die Effektivität des EXI Formates hinsichtlich Kompaktheit steht und fällt dabei mit der Definition eines entsprechend treffendem und zweckgebundenem XML Schemas. So ist der Gebrauch der in EXI eingebauten Datentypen, wie **Integer** oder **Float** etc., jederzeit dem ordinären Datentyp **String** vorzuziehen, da dieser durch seiner byte-weisen Repräsentation unverhältnismäßig mehr Speicher benötigt. Ebenso führt die Vermeidung optionaler Attribute durch Setzen der Parameter `use="required"` zu kürzeren Eventcodes zwischen den einzelnen States, da dadurch eine Mehrfachverzweigung verhindert wird.

Eine weitere Optimierung der Kompaktheit des EXI Streams wird durch Verwendung der Codierung im strict-mode erreicht. Hier entfällt bei der Auswertung der Eventcodes jeweils der State **Undeclared**, welches in vielen Fällen ebenfalls zu einer verkürzten Bitlänge derer führt. Obwohl in der Implementierung des Code-Generators dieser Modus bereits berücksichtigt wurde, konnte bis zum Abschluss dieser Projektarbeit keine korrekte Arbarbeitung derart codierter Streams gewährleistet werden.

# Literaturverzeichnis

- [1] *EXIficient - XML becomes efficient ...* <http://exificient.sourceforge.net/>, 2010.
- [2] Ferhat Ayaz. *XML und XSLT mit C++*. REDLINE GmbH, Heidelberg, 2006.
- [3] Zach Shelby Daniel Peintner, Mika Luimula. *Efficient XML Encoding and 6LowApp*. <http://tools.ietf.org/id/draft-shelby-6lowapp-encoding-00.html>, 2009.
- [4] Elliotte Rusty Harold & W. Scott Means. *XML In A Nutshell*. O'Reilly Verlag GmbH & Co. KG, 2005.
- [5] W3C. *XML Schema Part 2: Datatypes Second Edition*. <http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/>, 2004.
- [6] W3C. *XML Binary Characterization*. <http://www.w3.org/TR/xbc-characterization>, 2005.
- [7] W3C. *Analysis of the EXI Measurements*. <http://www.w3.org/XML/EXI/report.html>, 2006.
- [8] W3C. *Efficient XML Interchange (EXI) Format 1.0*. <http://www.w3.org/TR/exi/>, 2009.
- [9] W3C. *Efficient XML Interchange (EXI) Primer*. <http://www.w3.org/TR/2009/WD-exi-primer-20091208/>, 2009.

## **8 Erklärung über die selbständige Anfertigung der Arbeit**

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne unerlaubte Hilfe Dritter verfasst sowie keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Angaben, die inhaltlich oder wörtlich aus fremden Werken stammen, wurden kenntlich gemacht. Diese Arbeit lag in gleicher oder ähnlicher Weise noch keiner Prüfungsbehörde, Fachhoch- oder Hochschule vor und wurde ebenfalls bisher noch nicht veröffentlicht.

Rostock, 13.12.2010

Maik Gotzmann